

The role of aesthetics in understanding source code

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

ED120 - THALIM

last updated: 2023-08-14

Abstract

This thesis investigates how the aesthetic properties of source code enable the representation of programmed semantic spaces, in relation with the function and understanding of computer processes. By examining program texts and the discourses around it, we highlight how source code aesthetics are both dependent on the context in which they are written, and contingent to other literary, architectural, and mathematical aesthetics, depending on different scales of reading. Particularly, we show how the aesthetic properties of source code manifest expressive power due to their existence as a dynamic, functional, and shared computational interface to the world, through formal organizations which facilitate semantic compression and spatial exploration.

Contents

1	Introduction	5
1.1	Context	7
1.1.1	The research territory: code	7
1.1.2	Beautiful code	12
1.1.3	Literature review	18
1.2	The aesthetic specificities of source code	34
1.2.1	What does source code have to say about itself?	36
1.2.2	How does source code relate to other aesthetic fields?	37
1.2.3	How do the aesthetics of source code relate to its functionality?	38
1.3	Methodology	38
1.4	Roadmap	42
1.5	Implications and readership	45
2	Aesthetic ideals in programming practices	48
2.1	The practices of programmers	49
2.1.1	Software developers	50
2.1.2	Hackers	66
2.1.3	Scientists	81
2.1.4	Poets	95
2.2	Ideals of beauty	109
2.2.1	Introduction to the Methodology	109

2.2.2	Lexical Field in Programmer Discourse	113
2.3	Aesthetic domains	131
2.3.1	Literary Beauty	132
2.3.2	Scientific beauty	143
2.3.3	Architectural beauty	148
3	Understanding source code	161
3.1	Formal and contextual understandings	163
3.1.1	Between formal and informal	164
3.1.2	Knowing-what and knowing-how	172
3.2	Understanding computation	181
3.2.1	Software ontology	181
3.2.2	Software complexity	189
3.2.3	The psychology of programming	200
3.3	Means of understanding	207
3.3.1	Metaphors in computation	208
3.3.2	Tools as a cognitive extension	218
4	Beauty and understanding	229
4.1	Aesthetics and cognition	230
4.1.1	Source code as a language of art	231
4.1.2	Contemporary approaches to art and cognition . . .	241
4.2	Literature and understanding	246
4.2.1	Literary metaphors	246
4.2.2	Literature and cognitive structures	251
4.2.3	Words in space	257
4.3	Architecture and understanding	267
4.3.1	Form and Function	268
4.3.2	Patterns and structures	274
4.3.3	Material knowledge	290
4.4	Forms of scientific activity	295

4.4.1	Beauty in mathematics	296
4.4.2	Epistemic value of aesthetics	302
4.4.3	Aesthetics as heuristics	309
5	Machine languages	319
5.1	Linguistic interfaces	320
5.1.1	Programming languages	321
5.1.2	Qualities of programming languages	330
5.1.3	Styles and idioms in programming	348
5.2	Cognitive aesthetics in program texts	356
5.2.1	Between humans and machines	357
5.2.2	Matters of scale	360
5.2.3	Semantic layers	374
5.3	Functions and aesthetics in source code	388
5.3.1	Functional beauty	389
5.3.2	Functions of source code	394
5.3.3	Aesthetic and ethical value in program texts	401
6	Conclusion	407
6.1	Findings	408
6.1.1	What does source code have to say about itself?	409
6.1.2	How does source code relate to other aesthetic fields?	414
6.1.3	How do the aesthetics of source code relate to its functionality?	419
6.2	Contribution	421
6.2.1	Limitations	426
6.3	Opening	427

To me, programming is more than an
important practical art.

It is also a gigantic undertaking in the
foundations of knowledge.

Grace Hopper

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 functions that such configurations fulfill with regards to understanding. This inquiry will lead us to consider the different ways in which source code can be represented, depending on its aims 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 software, this thesis focuses on studying the reality of written code, along with its conceptual interpretations.

Starting from concrete instances 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, that source code is a medium for expression, both to express the programmer's intent to the computer (Dijkstra, 1982) and

the programmer's intent to another programmer (Abelson et al., 1979)—throughout this study, 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, clay or natural language. As such, the development and solidification of aesthetic practices—that is, of ways of doing which focus on the presentation on an artefact at least as much as on its function—is an ongoing research project in computer science, software development and the digital humanities (see our literature review in 1.1.3). 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 can thus be written in a way that 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 aesthetic domains to assess source code, as a means to grasp the artefact that is software. These draw on metaphors ranging from literature, architecture, mathematics and engineering. And yet, source code, while related to all of these, isn't exactly any of 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 situates itself.

The examination of source code, and of the discourses around source code will integrate both the variety 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 practitioners tend to deploy references to conceptual metaphors drawing from the domains above, but also how these references overlap across groups. The point of overlap, as we will demonstrate, is that of *using a formal linguistic system to communicate the understanding of complex cognitive structures, at the*

interface of the computational and of the natural. Through an interdisciplinary approach, we will 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 address 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 storage, communication and creation of information which support each of these domains of human activities. These complex processes are described in source code, a vast and mostly invisible set of texts. The number of lines of code involved in supporting human activity 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)

```

a = 4
b = 6

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

compute(a, b)

```

Listing 1: Example of the basic elements of a computer program, written in Python

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 looking more closely at 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, often resulting in mechanical action (e.g. a change in movement, display or sound). For instance, using the language called Python, the source code in 1 consists in telling the computer to store two numbers in what are called *variables* (here, *a* and *b*), then proceeds with describing the *procedure* for adding the double of the first terms to the second term (here, *compute*), 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 symbolize the successive states that the com-

puter 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 the existence or appearance of 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. Programmers are the ones who write the source code and this process is first and foremost a collaborative endeavour. They 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, requirements, 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 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). (Linux Information Project, 2006). 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 3 (*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 inter-subjective process (Voloshinov & Bachtin, 1986), and as such is different from the binary representation of a program, an object which we do not consider fitting to the frame of our study due to its unilaterality—among computers and humans, only humans can effectively read it. The intelligibility of source code, they continue, facilitates its circulation and duplica-

tion among programmers. It is this aspect of a socio-technical object that we consider as important as its procedural effectiveness.

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 successfully actionable response from both a computer and a human, 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. Similarly, the recent development of large language models in deep learning have ushered a new kind of source: a well-formed statistical representation of source code, aggregated from various sources into an answer to a specific problems. While these do pose inter-

esting questions in terms of intentionality and style, we nonetheless also reserve this kind of source code to a subsequent study.

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, 2001), and that of a open-ended, multi-layered document, in the vein of Barthes' distinction between a text and a work (Barthes, 1984). We will refer to the general medium of a textual interface to computation as source code, and to the coherent, practical instance of software manifested through source code as program texts.

1.1.2 Beautiful code

From 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 translation tasks from thought to machine, and which required more mechanical than notational 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, ad-

dresses directly both the existence of programming as an activity separate from mathematics and engineering, as well as an activity with an "artistic" dimension.

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 2.3.3.

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 more often through practice than 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 and functional purposes rather than strictly quantitative measurements (Pye, 2008). Approaching programming as a craft has been a recurrent perspective (Lévy, 1992; Dijkstra, 1982), and 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 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.

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, and is still discussed today. 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 philosophical inquiries in the Western tradition, from beauty as the imitation of nature, moral purification, disinterested appreciation, cognitive perfection, or sensible representations with emotional repercussions. 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 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, resembling, 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, according to him, 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.

Goodman's 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.

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, which he calls *the art of language*, results in the establishment of two dichotomies: fiction/diction, and constitutivity/condi-

tionality. 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-existing structures, themes and genres. Focusing on conditionality, this approach paves the way for an extension 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, through its execution. Because source code always holds software as a potential within its markings, 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—on what it does, or on how it does it. 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 normative definitions of being aesthetically pleasing, or because the software itself is considered a piece of art in the socio-economic sense, 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 possibil-

ity 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 thus 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 only secondly on what it *does*¹. 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*, and reaching into the domain of *everyday aesthetics* (Saito, 2012).

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 with no external reference, but rather to acknowledge that there is a certain difference between the content of software and the form of its source—aesthetically pleasing source code does not guarantee great software. 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. Such physical manifestations can, in turn, support an evaluative appraisal of their objects of the concern, enabling an aesthetic judgment.

We also distinguish the aesthetic from the beautiful, which implies an

¹As we have seen with Goodman, there is nonetheless a tight connection between those two states.

emotional response and is closely tied to the artistic status of an artefact; we instead focus on the positive properties in everyday encounters, rather than in an art-historical context.

This overview of the theoretical frameworks of this thesis is already implicitly setting the boundaries of this study. The domain we are investigating here is one that is delimited by both medium and purpose. First, the medium limitation 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 digital affordances (e.g. graphical user interfaces (Gelernter, 1998), real-time interactivity (Laurel, 1993) and process-intensive developments (Murray, 1998)) differ from the literary and architectural ones that software, in its written form, is claimed to exhibit. However, executable and executed software, being two sides of the same coin, might suggest causal relationships—e.g. the aesthetics of source code affecting the aesthetics of software—but we reserve such an inquiry for a subsequent study.

Now that we have explicated our object of study—the formal manifestations of software under its textual form—we can turn to a review of the research that has already been done on the subject, before highlighting some of the current limitations.

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 Dijkstra regarding the importance of paying attention to the aspects of programming practice (Dijkstra, 1972) which go beyond strictly mathematical and engineering requirements, Kernighan 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 program in 2 can be rewritten into the program in 3, which keeps the exact same functionality, but exhibits different formal manifestations. Specifically, the first listing involves a special case for each single digit, while the second listing reduces the syntax, allowing the program text to gain not just in concision but also in generality.

Why it becomes much clearer, though is not explicated by the authors in terms of concepts such as cognitive surface, repleteness of a symbol system or metaphorical 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 depth of 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 role 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

```

void letpad(int i)
{
    char* c;
    if (i == 0)
        c = "00";
    if (i == 1)
        c = "01";
    if (i == 2)
        c = "02";
    if (i == 3)
        c = "03";
    if (i == 4)
        c = "04";
    if (i == 5)
        c = "05";
    if (i == 6)
        c = "06";
    if (i == 7)
        c = "07";
    if (i == 8)
        c = "08";
    if (i == 9)
        c = "09";
}

```

Listing 2: A very verbose way to left pad a digit with zeroes in the C language.

```

void letpad(int i)
{
    char *c;
    if (i >= 0 && i < 10)
        c = '0' + i;
}

```

Listing 3: A very terse way to left pad a digit with zeroes in the C language.

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). This is an identification of symptoms, but without explicit connection to a possibly common cause.

In the context of formal academic research, such as the IEEE or the Association for Computing Machinery (ACM), subsequent work focuses on how to quantitatively assess a given quality of source code either through a social perspective on stylistic stances (Oman & Cook, 1990a), on the process of writing (Norick et al., 2010), a semantic perspective on the lexicon being used (Fakhouri 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) or through direct interviews (Hermans et al., 2020). These focus on the connection of aesthetics with the performance of software development—beautiful code as being related to a productive programmer and 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. In the emerging field of the psychology of aesthetics, we can point to the work of Kozbelt, Dexter et. al., who conducted quantitative surveys of programmers' relationship with aesthetics (Kozbelt et al., 2012), as well as qualitative analyses of the relationship between embodiment, aesthetics and code (Dexter et al., 2011). The latter study also investigated the metaphorical references that programmers make to code, showing how programmers use terms like *flow*, *balance*, *flexible* to refer to beautiful code (Dexter et al., 2011). The parallel they establish between lexical uses and embodied cognition also draws on the work of Lakoff et. al. to consider these metaphors as having a cogni-

tive purpose, a methodology we also follow. This research aims to build on their research and develop, from their discussion of metaphor and embodiment, how we can conceive an aesthetics of source code with a relationship to various understandings of space.

The development of software engineering as a profession has led to the publication of several books of specialized literature, taking a more practical approach to writing good code. 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 the formal quality of code by relying on different traditions, such as the practical handbook, to explain why the formal aspect of code 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 of these studies is often empirical, in the case of academic articles, looking at large corpora or interviewing programmers in order to draw conclusions regarding this relationship between formal properties and quality. 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 not, however, include a comprehensive and consistent 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*, in which he develops a personal perspective based on experience. 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 with distinct practices and standards (Hayes, 2017) (see 2.1). None of these studies considers whether the conclusions established for one group would be valid for the others.

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 well as in their demonstration 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 and focusing on its relationship to patterns, 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 human-

ities or artistic point of view than form a strictly scientific 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 mechanical, and suggest that additional 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 as primary sources, and are considered as texts to be read, examined and interpreted.

A first look at *Aesthetic Computing*, edited by Paul A. Fishwick allows us to highlight 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, 2006). 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, it 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 preferred lens. 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, second-hand account, rather than formal, definition of a source code aesthetic through the close reading of program texts.

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 poetry as an overtly literary use of code, even though it represents only a minor fraction of all program texts. 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 scholars 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, discrete/continuous, compilation/interpretation, and flat/stacked languages, clearly acting as a 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 fruitful 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 its structure and therefore provides more insight as to how humans represent code through speech, rather than representing speech through code. This presents a somewhat postmodern view of programming languages, approaching them as a relational, mutable conception of language as a 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 semantic object with a political load affecting the multiple economic, social 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, and focusing on its imperative qualities. 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 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 attention to source code into an older web of historical attempts at integrating combinatorial and supernatural 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). Such a positioning of technology across the realms of art, religion and knowledge can also be found in 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 discrete forms which relate to the totality of the world. As 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 research, given our particular attention to textuality. Yet, we will see how how themes of obfuscation, fragmentation, language and material will come into play as we develop our inquiry. 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 such, it highlights that, code does have social components of varying natures, insofar as it operates as an expressive medium between varying subjects.

Adrian MacKenzie takes on such a social approach to 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 code-based artworks and Alex McLean's `forkbomb.pl` (see 81), 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 highly-constrained 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 organizational processes *around* code (e.g. work management, specifications, test suites), rather than on the processes *within* code (e.g. styles, files and 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 also applies, as we will see in 5.2, to those intrinsic properties of source code.

Focusing specifically on the category of code poetry, Camille Paloque-Bergès published, a couple of years later, *Poétique des codes sur le réseau in-*

formatique (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. The first chapter focuses on digital literature as the result of executed code in order to develop a heuristic to approach source code, while 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. In the second chapter, PaloqueBergès provides an introduction to 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 instances of poetic program texts.

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 a rigorous close-reading of source code, in a deductive fashion, working from the words on the screen and elaborating on the context within which these words exist, in order to establish the cultural relevance of source code. While the study itself, being a close-reading of a single 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 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 have seen in Cox and McLean's code, or in Paloqué-Bergès's examples, 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, assuming that 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 repurposing 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 continuation of 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, but does not however address the nature of the processes into 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—as well as with the problem domain which code aims at depicting.

This activity of programming as craft, already acknowledged by programmers themselves, is further explored in Erik Piñeiro's doctoral thesis (Piñeiro, 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 practitioners. 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 *instrumental 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 programmers actually say, as they exchange about their practice online, he uncovers some aesthetic ideals underpinning a certain practice of programming. However, there remains two limitations: it is not clear how source code as a 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 code poets, hackers or scientists. Nonetheless, this empirical approach from the discourses of programmers is a methodology which this study shares.

In summary, 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, and from a humanities perspective.

In the former approach, aesthetics are acknowledged as a component of reading and writing code, and assessed through practical examples, quantitative 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 starting point of varied, empirical investigations, but do not consider how source code aesthetics might overlap with various other aesthetic domains.

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 equated with a new kind of writing. There is a potential for beauty and art in source code, particularly salient in code poetry, but such potential is not assessed through the same empirical lense as the former part of our literature review, even though it also addresses which intrinsic features of code can support aesthetic judgments and experiences.

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 de-

vice, yet it remains unclear in 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, as 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. These 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 referred to when assessing aesthetics in source code. By aesthetic field, we mean the set of medium-specific symbol systems which operate coherently on a stylistic and 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 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 4, the first gap to highlight here is how these multiple aesthetic fields are used to frame the aesthetics of source code, without this plurality being explicitly 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 how each of these map to various aspects of source code.

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. 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, is rarely confronted with the concrete, physical manifestations of computation in the form of source code.

In consequence, there are theoretical frameworks that emerge to explain software (e.g. computation, procedurality, protocol), but no comprehensive frameworks 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 through a dialectical approach. In the case of the particular object of this study, the establishment of such framework taking into account both the specifically textual dimension of source code and the various practices of all sorts of programmers 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 assuming their autotelic nature as art-objects.

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

lated 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 environments, inscribing it within platform studies. 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 has not 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.

Following this 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 research questions.

1.2.1 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 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, abstraction layers, function signatures, data types, variable declaration and variable nam-

ing, 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 socio-technical 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.

1.2.2 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 *elegant*, *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 code towards a more specific one, as the meeting point for all these fields, source code might reveal deeper connections between each of those.

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

The final 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 *contra* to the way of a rather traditional opposition between beauty and functionality, and therefore suggests 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 function, therefore relating back to Goodman's concept of the languages of art, of which programming languages can be part of. Particularly, this study will investigate how aesthetic configurations aim at making complex concepts understandable.

1.3 Methodology

To address such questions, we propose to proceed from 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.

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. They range from a few lines to several thousands, date between 1969 and 2021, with a majority written by authors in Northern 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 system². Both can be written in a variety of programming languages, as long as these languages are composed in unicode-encoded 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 exhibited 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. We will make use of this technique in order to illustrate some of our points. Rather than discussing complex code snippets, we will sometimes list translated, simplified versions in the Python programming language, and refer to the reader to the actual listings in the annex. 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 support 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 exclusively constitute program texts (unless they are considered strictly in

²A build system is a series of code transformations intended to generate executable code.

their implementation details as lexers, interpreters and compilers, themselves described in program texts), but are a necessary 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. Programming languages are hybrid artefacts, and their intrinsic qualities are only assessed insofar as they relate to the aesthetic 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, explanation and justification for a given extract of source code, and is a significant part of the software ecosystem. Even though they are written in natural language, 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)³. Examples of external commentaries include user manuals, textbooks, documentation, journal articles, forums discussions, 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 unfamiliar linguistic or hardware environments, 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 than that of the whole, our interpretation of meta-texts will take place via

³Such a distinction isn't a strict binary, and systems of inscription exist which couple code a commentary more tightly, such as WEB or Jupyter Notebook.

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 this aim, we will also mobilize metaphor theory from Lakoff to identify some of the properties of code as a target domain through some of the features of the aesthetic fields taken as source domains (Lakoff & Johnson, 1980).

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 Mary-Laure Ryan, 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. Mathematical beauty will be considered in its capacity to communicate complex concepts as well

as to act as a heuristic when developing proofs for complex theorems, as explicated by scholars such as Gian-Carlo Rota and Nathalie Sinclair. Throughout, we will see how an approach to craft, as the enactment of tacit knowledge in the creation of functional artefact can apply these domains.

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 2, in this study is an empirical assessment of how programmers consider aesthetics with their practice or reading and writing it. 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—elegance, 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 show how the overlap of these concepts can be found in the process of *understanding*—communicating abstract ideas through concrete manifestations. Indeed, we will see that *how* source code is written allows us to grasp *what* it does.

After establishing the role of aesthetics as the answer to source code's cognitive complexity, we will proceed to analyze further such a relationship between understanding, source code and aesthetics in 3. 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. This ambivalent status will be developed through the notion of *abstract artifact*, highlighting both material and cognitive dimensions of our object of study. We will show how source code needs to 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. In particular, one of the ways that enable humans to grasp computational concepts are metaphorical devices. 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.

Taking a step back, we will then assess in 4 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 to suggest a particular grasp on concepts of time and space. 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 a text to be understood, and on how aesthetics can enable understanding, we turn to its ef-

fective manifestations to develop our framework in 5. First, we will see how programming languages act as linguistic interfaces to computational phenomena, both from an objective and from a subjective perspective. Considering 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 valuable code could still be based around concepts of effectiveness, simplicity and performance. Human use of programming languages reaches into the extreme of *esolangs*—an investigation into those will reveal that language can be considered as a material, one whose base elements can be recombined to represent specific structures. 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 experience differ across linguistic communities of practice, and where they overlap, tracing connections between specific textual configurations of source code with the ideals summoned by the programmers. Finally, we will conclude on how aesthetics are both conditioned to the function of the artefacts they are manifested in, and themselves perform a functional role in epistemological communication, operating through metaphorical references and structural arrangements at various scales.

We will then turn back to our research questions to show how semantic compression and spatial exploration are crucial components of source code aesthetics. 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, intuitive understanding of the problem domain, and ability to shift perspectives) and of the machine (such as speed of execution, and reliance on explicit formal grammars). This duality will also be seen in 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 ambigu-

ity. 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 an executed program text. We will show how particular formal configurations, at the level of vocabulary, syntax, structure and style, ultimately involve the compression of human semantics and computer semantics, in conjunction with the ability to enable non-linear, writerly exploration of the program texts.

Finally, we will relate Goodman's conception of art as cognitively effective symbol system, 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 conceptions 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 we mean when we refer to *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 and the discourses around them, 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 aesthetics play a communicative role, and how concrete manifestations can, through a

metaphorical process, hint at broader effective 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 context-sensitivity of aesthetic standards: practices, uses and purposes determine as much, if not more, of the aesthetic value of a given program text, than a shared medium.

By examining the object 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 it is their use which 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 will clarify what this essential component of algorithms is, and opens up potential for further work in terms of thinking no longer of the aesthetics of source code, but of its poetics; that is, in the way source code, as a language of art, can also be 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 an activity which involves a creative process 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 how they approach the communicative function of their work as an aesthetic endeavour.

Finally, such a study of aesthetics, 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 means 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 functional 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 massively-distributed codebases to *ad hoc*, one-line solutions, cryptic puzzles and printed code.

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 that make up programmers' discourses, in which they qualify their practice. Out of these, we identify a cluster of adjectives and comparisons which will provide an empirical basis for considering the desirable and undesirable aesthetic properties of source code.

We 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 scientific and architectural conceptions of beauty can overlap and be referred to in the same medium. Such an overlap will reveal the importance of function, craft and knowledge in the disposition and representation of code. Our conclusion focuses on how understanding plays a central role in an aesthetic approach to source code, and results from the specificity of code as a cognitive material.

2.1 The practices of programmers

The history of software development is that of a practice born in the aftermath of the second world war, one 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 (Cohendet et al., 2001) and communities of practice (Hayes, 2017), in turn producing different types of source code. Each of these write source code with particular characteristics, and with different priorities in how knowledge is produced, stored, exchanged, transmitted and retrieved. In this section, 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 program texts 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 and playful constraints. *Scientists* operate within an academic environment, focusing on 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. Finally, *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-programmers¹.

2.1.1 Software developers

As Niklaus Wirth puts it, *the history of software is the history of growth in complexity* (Wirth, 2008), while also following a constant lowering of the

¹For instance, Microsoft's Visual Basic for Applications, Ableton's Max For Live, MIT's Scratch 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.

barrier to entry to the tools through which this complexity is managed. As computers' technical abilities in memory storage and processing power increased year on year since the 1950s, the nature of writing computer instructions shifted as well.

From machine dependence to autonomous language

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). Starting in the 1950s, and continuing throughout the 1960s, software developers were contractors hired to work directly with a specific hardware. These mainframes were large, expensive, and rigid machines, requiring platform-specific knowledge of the corresponding Assembly instruction set, the only programming language available at the time². 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 hardware-dependent³.

²One of the first operating systems, MIT's Tape Director, would be only developed in 1956 (Ross, 1986), which would facilitate some of the more basic memory allocation, process management, and system calls

³*But most important of all, the programmer himself had a very modest view of his own work: his work derived all its significance from the existence of that wonderful machine. Because that was a unique machine, he knew only too well that his programs had only local significance and also, because it was patently obvious that this machine would have a limited lifetime, he knew that very little of his work would have a lasting value. Finally, there is yet another circumstance that had a profound influence on the programmer's attitude to his work: on the one hand, besides being unreliable, his machine was usually too slow and its memory was usually too small, i.e. he was faced with a pinching shoe, while on the other hand its usually somewhat queer order code would cater for the most unexpected constructions. And in those days many a clever*

In the 1960s, 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—and expressivity—expressing a program text in a high-level, English-like syntax, rather than Assembly instruction codes. Programmers are no longer tied to a specific machine, and therefore acquire a certain autonomy, a recognition which culminates in the naming of the field of *software engineering* (Randell, 1996).

Campbell-Kelly concludes on a wave 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 the Commodore 64, Altair and Apple II, software could be effectively entirely decoupled from hardware. The writing 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). And yet, software immediately enters a crisis, where projects run over time and budget, prove to be unreliable in production and unmaintainable in the long-run. It is at this time that discussions around best practices in writing source code started to emerge.

This need for a more formal approach to the actual process of programming found one of its most important manifestations in Edsger Dijkstra'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 pro-

programmer derived an immense intellectual satisfaction from the cunning tricks by means of which he contrived to squeeze the impossible into the constraints of his equipment. (Dijkstra, 2007)

gram construction. Despite its laconic section titles⁴, Dijkstra'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 (Kay, 1993)) 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 justify 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), a magical solution addressing the cascade of potential risks which emerge from large software applications⁵. 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 solu-

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

⁵For instance, the *Forum on Risks to the Public in Computers and Related Systems* serves as a publication to centralize such concerns (Neumann, 1985)

tion, 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 wave in the history of software developers is the popularization of the Internet and of the World Wide Web, a network which was only standardized in 1982 and access to it was provided commercially in 1989. Built on top of the Internet, it popularized global information exchange, including technical resources to read and write code. Software could now be written on cloud computing platforms, shared through public repositories and deployed via containers with a lower barrier to entry than at a time of source code printed in magazines, of overnight batch processing and of non-time-sharing systems.

Engineering texts

Software developers have written some of the largest program texts to this date. However, due to its close ties to commercial distributors, source code written in this context often falls under the umbrella of proprietary software, thus made unavailable to the public. The program texts written by software developers are large, they often feature multiple programming languages and are highly structured and standardized: each file follows a pre-established convention in programming style, which supports 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, usability and attention to function.

The IEEE's Software Engineering Body of Knowledge (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, construc-

tion, testing and maintenance. 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. This essential step in software development aims at formalizing as best as possible the elements that exist beyond software, in order to make those computable, and the design of an adequate formalism is a fundamental requirement for a successful software application.

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 software architecture. Software architecture operates both from a top-down perspective, laying down an abstract blueprint for the implementation of a system and dictating how a program text is structured, how its parts interact, why it's built that way, consisting different components of an existing system interact (Brown & Wilson, 2011).

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 complexity⁶, in anticipation of likely changes and possible reuse by other software systems. Here, the empha-

⁶Complexity does not exist only at the programming level, but also at the architecture level: "*there are two ways of 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.*" (Hoare, 1981)

sis 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: the creation of working software through a combination of coding, verification, unit testing, integration testing, and debugging. (Bourque & Fairley, 2014). The practice of software engineers thus implements functional and reliable mechanical designs through, ultimately, the act of writing in formal languages.

Software maintenance, finally, relates not to the planning or writing of software, but to its reading. Software is notoriously filled with bugs⁷ which can be fixed through the release of software updates. This means that the life of a software doesn't stop when its first version is written, but rather when it does not run anywhere anymore: it can still 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 final component of software development directs us back to its notorious cognitive complexity, one that increases with the age of the software.

What does this look like in practice? In order to understand the aesthetic preferences of software developers, we must start by assessing the kinds of program texts they write. We look at excerpts from two code bases: the source code for Microsoft Windows XP, which was started in 2001 (Warren, 2020), and the Kirby CMS project, started in 2011; the quantitative specificities of both code bases are shown in 2.1. While these two projects differ drastically in their size, in their age, and in the number of developers involved in their creation and maintenance, we nonetheless choose them as the respective ends of a single spectrum of software engineering. In both cases, the prime concern is with function and maintain-

⁷McConnell estimates that the industry average is about 15 - 50 errors per 1000 lines of delivered code. (McConnell, 2004).

	Microsoft XP	Kirby CMS
number of files	28,655	1,859
total lines	13,468,327	345,793
number of file extensions	10	6

Figure 2.1: Table comparing the scale of two software development projects.

ability.

First, the most striking visual feature of the code is its sheer size. In the case of Microsoft XP, representing such a versatile and low-level application such as an operating system results 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 on. Looking at the file `cmdatini.c`, reproduced partially in 4, we see very long variable names, with a rhythmic, repetitive structure where differences between lines is not obvious at first.

The repetition of the `RtlInitUnicodeString` in the first part of this listing stands at odds with the second part of the code, the `for()` statement, displaying a contrast between between a verbose text and a compressed text. Verbosity, the act of explicitly writing out statements which could be functionally equivalent in a compacted form, is a feature of the Windows 2000 codebase, one which is a consequence of a particular problem domain, of a particular business imperative of maintainability, and of the particular semantic environment of the C programming language.

The problem domain of the Windows XP operating system, its longevity and its update cycle, all contribute to its complexity and have affected how this code is written. Here, the problem domain of the program text is the computer hardware, and its function is to make sure the kernel knows about the hardware it is running on (e.g. its name, its description, etc.), in

```

{
    ULONG i;

    RtlInitUnicodeString(&CmRegistryRootName,
                        CmpRegistryRootString);

    RtlInitUnicodeString(&CmRegistryMachineName,
                        CmpRegistryMachineString);

    RtlInitUnicodeString(&CmRegistryMachineHardwareName,
                        CmpRegistryMachineHardwareString);

    RtlInitUnicodeString(&CmRegistryMachineHardwareDescriptionName,
                        CmpRegistryMachineHardwareDescriptionString);

    RtlInitUnicodeString(&CmRegistryMachineHardwareDescriptionSystemN
    ↵ ame,
    ↵             CmpRegistryMachineHardwareDescriptionSystemS
    ↵ tring);

    RtlInitUnicodeString(&CmRegistryMachineHardwareDeviceMapName,
                        CmpRegistryMachineHardwareDeviceMapString);

    RtlInitUnicodeString(&CmRegistryMachineHardwareResourceMapName,
                        CmpRegistryMachineHardwareResourceMapString);

// ...

// 
// Initialize the type names for the hardware tree.
//

for (i = 0; i <= MaximumType; i++)
{
    RtlInitUnicodeString(&(CmTypeName[i]),
                        CmTypeString[i]);
}

// ...
return;
}

```

Listing 4: Unicode string initialization in Microsoft 2000 operating system, with a first part showing an explicit repeating pattern, while the second part shows a more compressed approach.

an explicit and verbose way, before more compressed writing techniques can be used. Dealing with a specific problem domain (i.e. kernel instructions) leads to a specific kind of aesthetics; here, forcing the programmers to repeat references to `RtlInitUnicodeString()` 1580 times across 336 files.

Another significant aesthetic feature of the Windows 2000 program text is its use of comments, and how those comments point to a collaborative, layered authorship. This particular program text is written across individuals and across time, each with presumably its own approach. Yet, writing source code within a formal organization often implies the harmonization of individual approaches, and thus 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). The excerpt in 5 from `jdhuff.c` is a example of such overlapping of styles.

Comments are specific lines of source code, identified by particular characters (in the C programming language, they are marked using `//` and `/* */`), which are ignored by the machine. That is, they are only expected to be read by other programmers, and in this case primarily by programmers belonging to a single business organization. Here, the variety of comment characters and the variety of capitalization hint 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.

Treated as natural language, comments are not procedurally guaranteed to be reflected in the execution, of the program, and are considered by some as misleading: they might be saying something, while the code does something else⁸. Beyond the presence of multiple authors, this excerpt also exemplifies the tension between source code as the canonical source of knowledge of what the program does and how it does it and comments as a more idiosyncratic dimension of all natural-language expressions of

⁸This has led to the argument that only the source code has epistemological value in a software project: "*the only document that describes your code completely and correctly is the code itself*" (Goodliffe, 2007)

```

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; // 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 5: Overlapping programming voices can be hinted at by different comment styles.

```

/*++

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 6: pnpenum.c shows the explicit traces of multiple authors collaborating on a single file over time.

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. This is the case in the use of comment headers, which locate a specific file within the greater architectural organization of the program text (see 6). This highlights the multiple authors and the evolution in time of the file: comments are the only manifestation of this layering of revisions which ultimately results in the "final" software⁹.

Ultimately, the Windows XP source code shows some of the components at stake in the program texts written by software developers: verbosity and compression, multi-auctoriality, and natural language writing

⁹The term "final" is in quotes, since the Windows 2000 source contains the mention BUGBUG 7436 times across 2263 files, a testament to the constant state of unfinishedness that software tends to remain in.

in the midst of formal languages. Still, as an operating system developed by one of the largest corporations in the world, it also possesses some specificities due to its problem domain, programming language and socio-economic environment.

Another example of a program text written by software developers, complementing Windows XP, is the Kirby CMS (Allgeier, 2022). With development starting in 2011 and a first release in 2012, it developed a steady community of users, shown in consistent forum posts and commit history on the main repository. Kirby is open-source, meaning that it affords direct engagement of other developers with its architecture through modification, extension or partial replacement, content management system. Its problem domain is therefore the organization of user-facing multimedia assets, such as text, images and videos.

The Kirby source code is entirely available online, and the following snippets hint at another set of formal values—conciseness, explicitness 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 1859 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 file¹⁰.

If we look at a typical function declaration within Kirby, we found one such as the `distinct()` setter for Kirby's database, reproduced in 7. This function allows the developer to set whether she only wants to select distinct fields in a database query.

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

¹⁰As a comparison, the leading project in the field, Wordpress.org, has 3466 files, with the longest file comprising 9353 lines of code (`customize-controls.js`), and the shortest file (such as `script-loader-packages.php`) (Wordpress, 2023)

```

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

```

Listing 7: The setting of whether a query should be distinct includes some verbose details which prove to be helpful in the long run (Allgeier, 2021b).

semantic task of deciding whether a query should be able to include distinct select clauses (as opposed to only allowing join clauses), is now decoupled from its actual implementation. 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 succinct in code. Taking from the `Http\\Route` class, reproduced in 8, we can see a different approach to comments than in 5 of Microsoft XP operating system.

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 in the form of a website or a printed document. This is noticeable 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

```

/***
 * 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 8: The inclusion of comments help guide a programmer through an open-source project (Allgeier, 2021c).

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. Once again, we see that the source of truth is the computer's ability of reading input and executing it.

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 8, or on generic meta-information, but rather natural language renderings of the thought process of the programmer. 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. The reason for those comments becomes clear: that the small core of the function is actually hard to understand.

Looking at some excerpts from the Kirby program texts, we see a small number of files, overall short file length, short function length, consistent natural language comments and concise functionality. These aesthetic features give an impression of building blocks: short, graspable, (re-)usable components are made available to the developer directly, as the open-source project relies on contributions from individuals who are not expected to have any other encounter with the project other than, at the bare minimum, the source code itself.

In conclusion, these two examples of program texts written by soft-

```
// fall back to little execCommand hack with a temporary textarea
const input = document.createElement("textarea");
input.value = value;
document.body.append(input);
```

Listing 9: Even in a productive and efficient open-source project, one can detect traces of "hacks" (Allgeier, 2021a).

ware developers, Microsoft Windows XP and Kirby CMS, show particular presentations of source code—such as 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, at least in its executed form.

Indeed, software developers are a large group of practitioners whose focus is on producing effective, reliable and sustainable software. This leads them to writing in a relatively codified manner. And yet, we must acknowledge that idiosyncrasies in source code emerge; in 9, a function handling text input uses a convoluted workaround to store text data. Even in business environments and functional tools, then, the hack is never too far. The boundary between groups of practitioners is not clear-cut, and so we now turn to the correlated practice of hackers.

2.1.2 Hackers

To hack, in the broadest sense, is to enthusiastically inquire about the possibilities of exploitation of technical systems¹¹. Computer hacking specifically came to proeminence as early computers started to become available

¹¹"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)

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 efficacy¹².

The histories of hacking and of software development are deeply intertwined: 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 a product at the intersection of corporate and hacker culture, built by a small team, circulating along more or less legal channels, and spreading its design philosophy of clear, modular, simple and transparent design across programming communities (Raymond, 2003).

Hacker culture built on this impetus to share source code, and hence to make written software understandable from its textual manifestation. As hardware stopped 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 process; around the 1980s, executable software was distributed directly as a binary file, its exact contents an unreadable series of os and 1s.

¹²A meritocratic stance which has been analyzed in further in (Coleman, 2018)

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 latter 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 stealth¹³. These practices nonetheless refer to the original conception of hacking—getting something done quickly, and well—and include such a practical, efficient approach into its own set of values and ideals. In turn, these are represented in the kinds of program texts being written by members of this community of practice.

Meanwhile, the open-source movement took the tenets of hacking culture and adapted it to make it more compatible to the requirements of businesses. Indeed, beyond the broad values of intellectual curiosity and

¹³For 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)

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)¹⁴.

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, the necessity of skill for writing such software, and a certain inclination towards "quick and dirty" solutions.

Program texts as puzzles

Incidentally, those political and ethical values of expertise and openness often overlap with aesthetic values informing how their code exists in its textual manifestation. By looking at a few program texts written by hackers, we will see how their skillful engagement with the machine, and their playful stances towards solving problems is also reflected in how they write source code.

To hack is, according to the dictionary, "to cut irregularly, with-

¹⁴Another overlap can be found in the address of the software corporate giant Meta's headquarters: 1, Hacker Way, Menlo Park, CA 94025, U.S.A.

out 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)

Weizenbaum's perspective is that of a computer scientist whose theoretical work can be achieved only through thought, pen and paper. As such, he looks down on hackers as experts who can get lost in technology for its own sake. Gabriella Coleman, in her anthropological study of hackers, highlights that they value both semantic ingenuity¹⁵ and technical wittiness(Coleman, 2012). Source code written by hackers can take multiple shapes, from one-liners, to machine language magic and subversion of best practices in crucial moments.

The *one-liner* is a piece of source code which fits on one line, and is usually interpreted immediately by the operating system. They are terse, concise, and eminently functional: they accomplish one task, and one task only. This binary requirement of efficiency finds a parallel in a different kind of one-liners, the jokes of stand-up comedy. In this context, the one-liner also exhibits the features of conciseness and impact, with the setup

¹⁵Hackers themselves tend to favor puns—the free software GNU project is a recursive acronym for *GNU's Not UNIX*.

```

#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 10: This program text selects all the lines from an input file which is longer than 6 characters in the C programming language. See the one-line alternative implementation in 11.

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 order to understand how compression occurs in program texts, we can look at the difference between 10 and 11. Both of these have the same functionality: they select all the lines of a given input file.

In 10, achieving this functionality using the C programming language takes 20 lines. The equivalent in the AWK scripting language takes a single line, a line which the author actually refers to in a comment in 10, presumably as a personal heuristic as he is writing the function. 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 says that it prints every line in which the third field is greater than 6, and is easier to read, even for non-expert programmers. The AWK one-liner is more effi-

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

Listing 11: This program text selects all the lines from an input file which is longer than 6 characters in the C programming language, in just one line of code. See the alternative implementation in 20 lines of code in 10.

cient, more understandable because it allows for less confusion while also reducing the amount of text necessary, and is therefore considered more beautiful.

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), and each of these tools can then be composed in order to produce complex results—a feature of programming languages we will discuss in 5.1.1. 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?

Writing the shortest of all programs does become a matter of skill and competition, coupled with a compulsion to reach the most syntactically compressed version¹⁶. This behaviour is also manifested in the practice of *code golf*, challenges in which programmers must solve problems by

¹⁶For instance, Guy Steele, an influential language designer, who worked on Scheme, ECMAScript and Java, among others, recalls: "*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)

```
life ← {>1 ω v.Λ 3 4 = +/ +≠ -1 0 1 ◦.⊖ -1 0 1 φ“ ⊜ω}
```

Listing 12: Conway’s Game of Life implemented in APL is a remarkable example of conciseness, at the expense of readability.

using the least possible amount of character¹⁷, or in contests such as the Mathematica One-Liner Competition (Carlson, 2010). 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.

On the other hand, however, one-liners can be so condensed that they loose all sense of clarity for a reader who does not have a deep knowledge of the language in which it is written, or of the problem being solved. For instance, 12 is an implementation of Conway’s game of life implemented in one line of the APL programming. Conway’s Game of Life is a well-known simulation where a small set of initial conditions and rules for evolution produce unexpected emergent complexity. Its combination with APL programming language, which makes an extensive use of symbolic graphical characters to denote functions and operations, leads to particularly dense and terse source code.

This particular example 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 tends to be seen as a display of the relationship between the programmer, the language and the machine, rather than between different programmers. On the other hand, the small nature of one-liners makes them highly portable and shareable. 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

¹⁷Here, the equivalent of *par* in golf would be the number of character used: the lower the number, the better.

```

float Q_rsqrt(float number)
{
    long i;
    float x2, y;
    const float threehalves = 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 * (threehalves - (x2 * y * y)); // 1st iteration
    ↵   // y = y * ( threehalves -
    ↵   ( x2 * y * y ) ); // 
    ↵   2nd iteration,
    ↵   // this can be removed

    return y;
}

```

Listing 13: This particular implementation of a function calculating the inverse square root of a number has become known in programming circles for both its speed and unscrutability.

input by hand, and during which access to computation wasn't widely distributed amongst society, being able to type just one line in a computer program, and resulting in unexpected graphical patterns created a sense of magic and wonder in first-time users¹⁸, surprised by how so little can do so much (Montfort et al., 2014).

Another quality of hacker code is the idiosyncratic solution to an intricate puzzle. The listing in 13 calculates the inverse square root of a given number, a routine but computationally expensive calculation need in computer graphics. It was found in the source code of id Software's *Quake* video game¹⁹.

What we see here is a combination of the understanding of the prob-

¹⁸The visual output of one of these one-liners can be seen at <https://www.youtube.com/watch?v=0yKwJJw6Abs>.

¹⁹The 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.

lem 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 comments²⁰. 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. Paradoxically, the author Greg Walsh displays a very deep knowledge of how IEEE standards represent floating point numbers, to the extent that he is able to bend such standards into productive edge cases. While it is obvious what the program text does, it is extremely difficult to understand how.

This playfulness at writing things that do not do what it seems like they do is another aspect of hacker culture. The Obfuscated C Code Contest, 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. Obfuscated code is a first foray into closely intertwining these separate meanings in the source code itself, making completely opaque what the code does, and inviting the reader to decipher it.

The source code in 14, submitted to the 1988 IOCCC²¹ 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 indenta-

²⁰what the fuck?, indeed.

²¹Source: <https://web.archive.org/web/20131022114748/http://www0.us.ioccc.org/1988/westley.c>

```

#define _ -F<00||--F-00--;
int F=00,00=00;main(){F_00();printf("%1.3f\n",4.*-F/00/00);}F_00()
{
    -----
    -----
    -----
    -----
    -----
    -----
    -----
    -----
    -----
    -----
}

```

Listing 14: westley.c, entry to the 1988 IOCCC

tion, and would probably be useless in any other context, but nonetheless represents another aspect of the *concept* behind the procedure described, not relying on traditional programming syntax²², but rather on an intuitive, human-specific understanding²³.

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 does²⁴. 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

²²For such a program, see for instance: <https://crypto.stanford.edu/pbc/notes/pi/code.html>

²³Concrete poetry also makes such a use of visual cues in traditional literary works.

²⁴Also known informally as the "Aha!" moment, crucial in puzzle design.

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.

Building on the fact that source code very often does not do what one thinks it does when executed, initiatives such as the Underhanded C Code contest have leaned to this tendency. In this contest, one "*must write C code that is as readable, clear, innocent and straightforward as possible, and yet it must fail to perform at its apparent function. To be more specific, it should perform some specific underhanded task that will not be detected by examining the source code.*" (Craver, 2015). Hackers find value in this kind of paradigm-shifting: if software developers spend time attempting to make faulty, complex code easy to grasp and reliable, hackers would rather spend effort and skill making faulty code look deliberately functional.

Such intimate knowledge of both the language and the machine can be found in the program texts 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, within a fixed technical constraint²⁵. Indeed, the

²⁵For an example, see *Elevated*, programmed by iq, for a total program size of 4 kilobytes:

`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. 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. However, this presentation format acknowledges a kind of individual, artistic feat, rather than a collaborative, explicit text which tends to be preferred by software developers.

<https://www.youtube.com/watch?v=jB0vBmiTr6o>, winner of the 1st place at the Breakpoint 2009 contest.

00:	01	08	0b	08	ff	d3	9e	32	32	32	35	00	00	00	19	41
10:	1c	d0	00	dc	00	00	11	d0	e0	0b	10	33	0e	61	90	f5
20:	07	00	ff	1f	14	41	d5	24	15	25	15	53	15	61	d5	29
30:	1b	0f	e6	13	e6	13	d0	02	e6	20	a9	61	85	1c	a7	20
40:	e0	3f	f0	08	90	0c	4e	11	d0	6c	fc	ff	a0	6d	84	22
50:	84	d7	4a	4b	1c	a8	a5	13	29	30	d0	02	c6	1c	e0	2f
60:	f0	11	b0	02	a2	02	c9	10	f0	09	8a	29	03	aa	b5	f3
70:	85	0a	2d	ab	00	b0	11	b7	22	b6	21	95	00	a5	13	4b
80:	0e	aa	cb	f8	86	cc	49	07	85	0b	a5	13	29	0f	d0	0f
90:	a9	b8	47	14	90	02	85	14	29	07	aa	b5	f7	85	12	a0
a0:	08	b7	0d	91	0f	88	10	f9	a8	b7	09	91	03	88	d0	f9
b0:	4c	7e	ea	78	8e	86	02	8e	21	d0	20	44	e5	a2	fd	bd
c0:	02	08	95	02	ca	d0	f8	8e	15	03	4c	cc	00	a9	50	8d
d0:	11	d0	58	ad	04	dc	a0	c3	0d	1c	d4	48	4b	04	a0	30
e0:	8c	18	d0	71	cb	e6	cb	71	cb	6a	05	20	a0	58	05	d5
f0:	91	cb	d0	df	2b	aa	02	62	00	18	26	20	12	24	13	10

Figure 2.2: The annotated representation of the compiled version of A Mind Is Born, a demo by Linus Åkesson. The different color overlays highlight the meaningful regions of the program (Akesson, 2017).

Pushing the boundaries of how much can be done in how little code, 2.2 shows a 256-bytes demo resulting in a minute-long music video (Akesson, 2017) on the Commodore 64. It is first listed as a hexadeciml dump by its author, without the original Assembly code²⁶.

As a display of knowledge, the author highlights how different hexadeciml notations represent different parts of the software. Along with knowledge of how hexadeciml instructions map to the instruction set of the specific chip of the Commodore 64 (in this case, the SID 8580), the practical use of these instructions takes productive advantage of ambiva-

²⁶The Assembly version of the source was subsequently re-assembled by J.B. Langston (Langston, 2017), for study purposes.

lence and side-effects²⁷.

Demosceners therefore tend to write beautiful, deliberate code which is hardly understandable by other programmers without explanation, and yet hand-optimized for the machine. In addition to software developers' attempts to make intelligible via their source code, this practice adds a perspective on the relationship between aesthetics and understanding. Here, aesthetics do not support and enable understanding, but rather become a proof of the mastery and skill involved in crafting such a concise input for such an overwhelming output; it hints that one needs a degree of expert knowledge in order to appreciate these kinds of program texts.

Hackers are therefore 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* and *technical context-sensitivity*. 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 char-

²⁷Linus Åkesson explains how he layers functionality on the same syntactical tokens: *We need to tell the VIC chip to look for the video matrix at address \$ocoo and the font at \$oooo. This is done by writing \$30 into the bank register (\$d018). 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.* (Åkesson, 2017)

acter or two. Throughout, there is the recurring requirement of doing the most with the least, of written parsimony leading to executed expansiveness.

Hacking therefore involves knowledge: knowledge of the hardware, knowledge of the programming language used and knowledge of the trade-offs acceptable all the while exhibiting an air of playfulness. They tend to *get the job done* and *do it for the sake of doing it*, at the expense of conceptual soundness. 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, they stand at the polar opposite of another community of source code practitioners. Scientists who write source code (of which computer scientists are a subset) engage with programming 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, 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 distinct ways, and as such open up the landscape of the type of code which can be written, as well as 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 see how the specific needs of usability, replicability and data structuring link back to standards of software development. Then, we will turn to the code written by computer scientists and examine how ideal of computation manifest themselves in concrete implementations.

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, aerodynamics and ballistics, among others (Oberkampf & Roy, 2010). 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 for faster and more correct processing. Beyond the military applications of early computer technology, the advent of computing technology would prove to be of great assistance in physics and engineering, as shown by Harlow and Fromm's article on *Computer Experiments in Fluid Dynamics*²⁸, or the report on *Man-Computer Symbiosis* by J.C.R. Licklider (Licklider, 1960).

The remaining issue is to make computers more accessible to scientists who did not have direct exposure to this new technology, and therefore might be unfamiliar to the intricacies of their use. While universities can afford mainframe computers so that scientists do not have to wait for the personal computer revolution, another vector for simplification and accessibility is the development of adequate programming languages. The intent is to provide non-computer scientists with easy means to instruct the computer on how to perform computations relevant to their work, ultimately aiming to situate computation as the third pillar of science, along with theorization and experimentation (Vardi, 2010).

Such an endeavour started with the BASIC²⁹ programming language. Developed in 1964 at Dartmouth College, it aimed at addressing this hurdle by designing "*the world's first user-friendly programming language*" (Brooks, 2019), and led the personal computer revolution by allowing non-

²⁸"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)

²⁹BASIC stands for Beginners' All-purpose Symbolic Instruction Code.

```

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 15: Mesh.m

technical individuals to write their own software. 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. It had also increased 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 friendly graphical user interface (GUI). Along with its powerful array-manipulation features, this ability to visualize large series of data and plot it on a display largely contributed to MATLAB's popularity (Moler & Little, 2020). The combination of 15 and 2.3 shows how concise the plotting of a three-dimensional plane is in MATLAB. In the source code, it requires only one call to `mesh`, and the output is a complete visual rendering, with reasonable and aesthetically pleasing visual default settings in the form of graded axes.

Along with MATLAB, Python represents the advent of the so-called scripting languages: programming languages 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 to interact directly with the computer hardware, and to constitute data structures

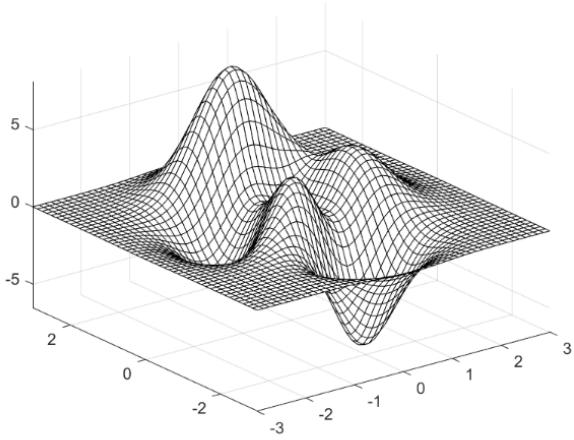


Figure 2.3: Visualization of a 3D-mesh in Matlab

from the ground up. 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`, as seen in 11) (Ousterhout, 1998). Starting with the late 1990s, and the appearance of languages such as Perl and Python, scripting languages became more widely used by non-programmers who already had data to work with and only needed the tools to exploiTThe development of additional scientific libraries such as *SciKit*, *NumPy* for mathematics and numerical work or *NLTK* for language processing and social sciences in Python complemented the language's ease of use by providing manipulation of complex scientific concepts (Millman & Aivazis, 2011).

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(Hatton & Roberts, 1994). Perhaps the most well-known instance of poor standards in programming was 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 software³⁰.

In response, 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, resulting in the formation of communities such as the Research Software Engineers(Woolston, 2022). As we have 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.

It seems that software's position in the scientific inquiry is no longer that of a helpful crutch, but rather of an inevitable step. 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 implementation and finally to assessment (Landau et al., 2011), code written as academic software is involved in the latter two stages of method and implementation. As such, it has to abide by the processes and requirements of scientific research. First and foremost, reproducibility is a core requirement of scientific research in general

³⁰Professor 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)

and bugs in a scientific software system can lead to radically different outputs given slightly different input data, while concealing the origin of this difference, and compromising the integrity of the research and of the researcher. 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 their own. Along with field-specific textbooks³¹ 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. Meanwhile, computer science worked towards asserting and pursuing its own field of research, sometimes distinct from the discipline of programming. Unlike other scientific fields possesses its own specific standards of programming, taking software not as a means to an end, but as the end itself.

³¹See *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.

Computation as an end

Computer scientists are scientists whose work focuses on computation as an object, 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, work on circuit and language design, work on mathematical foundations, information theory, systems theory and expert systems, computer science establishes its institutional grounding with the inauguration of the first dedicated academic department at Purdue University in 1962 (Ifrah, 2001).

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). Throughout the process of institutionalization, 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 (mathematical) properties, rather than its formal (aesthetic) properties. The first manifestations of such a theoretical fo-

cus can be found in the Information Processing Language in 1956 by Allen Newell, Cliff Shaw and Herbert Simon, which was originally designed and developed to prove Bertrand Russell's *Principia Mathematica* (Ifrah, 2001). While the IPL, as one of the very first programming languages, influenced the development of multiple subsequent languages, in particular some later languages came to be known as logic programming languages. These are based on a formal logic syntax of facts, rules and clauses about a given domain and whose correctness can be easily proven. We can see in 16 an example of the *Prolog* logic programming language. Its syntax appears very repetitive, a result of the few keywords used (`induce`, `element` and `clause`), and drawing directly from the lexical field of logic and framing the problem domain. Due to its Turing-completeness, one can write in Prolog programs such as language processing, web applications, cryptography or database programming, but its use seems to remain limited outside of theoretical circles in 2021, according to the Stackoverflow Developer survey for popular language uses (Overflow, 2021).

Lisp—*LIST Processor*—is another programming language which shares this feature of theoretical soundness faced with a limited range of actual use in production environments. It was developed in 1958, the year of the Dartmouth workshop on Artificial Intelligence, by its organizer, John McCarthy, and was designed to process lists. 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 that 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 is, processing something in terms of itself—Lisp highlights even further this tendency to separate computation from the problem domain, exhibiting autotelic tendencies. This is facilitated by its atomistic and relational structure: in order to solve what it has to do, it evaluates each symbol and traverses a tree-

```

% 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 16: The Prolog programming language focuses first and foremost on logic predicates in order to perform computation, rather than more practical system calls.

structure in order to find a terminal symbol. Building on these features of complex structures with simple elements, William Byrd, computer scientist at the University of Utah, describes the Scheme interpreter written in Scheme³² shown in (17) as "*the most beautiful program ever written*" (Byrd, 2017).

The beauty of such a program, for Byrd, is the ability of these fourteen lines of source code 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. Byrd equates the feelings he experiences in witnessing and pondering the program above to those suggested by Maxwell's equations, which constitute the founda-

³²Scheme is a Lisp dialect, designed a few years after Lisp itself, and also at MIT.

```

(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 17: Scheme interpreter written in Scheme, revealing the power and self-reference of the language.

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

Figure 2.4: Maxwell's equations form a terse, unified basis for electromagnetism, optics and electric circuitry.

tion of classical electromagnetism (see 2.4), a comparison that other computer scientists have made (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 relevant outputs—all the while allowing for complex and deep consequences for, respectively, computer science and electromagnetism.

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. 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, discrete 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. In this case, a restricted problem domain can enable a particular aesthetic judgment, but also exclude others.

"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 Dijkstra'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. Dijkstra 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, implicitly opposed, perspectives on programming. Program-

ming suffered from an earlier image of an inherently unmanageable, un-systematic, and artistic activity, 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 quantified 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 domains³³.

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 differs from it insofar as it does have to deal with concepts of effectiveness, implementation and contingency. 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).

Another influential textbook insisting that computation is not to be seen as an autotelic phenomenon is *Structure and Interpretation of Computer Programs*. In it, 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). Readability is thus an explicit standard in the discipline of

³³Notably, algorithms in textbooks tend to be erroneous when used in production; only in five out of twenty are they correct (Pattis, 1988).

```

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 18: Bubble Sort implementation in Julia uses the language features to use only a single iteration loop. (Moss, 2021a)

```

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

```

Listing 19: Nearest neighbor implementation in Julia (Moss, 2021b).

programming, along with a less visible focus on efficiency and verifiability. Finally, the aesthetic standard in this more engineering approach to computer science is, again, proportionality between the number of lines of code written and the complexity of the idea explained. We can see such a value at play in the series *Beautiful Julia Algorithms* (Moss, 2022). For instance, 18 implements the classic 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 details. The simplicity of scientific algorithms is expressed even further in 19 the one-line implementation of a procedure for finding a given element's nearest neighbor, a crucial component of classification 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, a concept we will explore further in 2.2.2.

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 $\tau\epsilon\chi\nu\eta$, 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, reability, sustainability, for instance 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. The beauty of scientific source code is thus associ-

ated with the beauty of other sciences, such as mathematics and engineering. And yet, Knuth is also known as the advocate of literate programming, a practice which engages first source code as a textual, rather than scientific, object. To address this nature, we complete our overview of code practitioners by turning to the software artists, who engage most directly with program texts through source code poetry.

2.1.4 Poets

Ever since Christopher Stratchey's love letters, programmers have been curious of the intertwining of language and computation. 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. Here, 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. Particularly, we pay attention to the role of function, correctness and meaning-making in these particular program texts.

Code poetry as executed literature

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 in 1953 with Christopher Stratchey's love letters, 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 semanti-

cally 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 code 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 (Montfort et al., 2014; Marino, 2020; Brock, 2019). 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 only written works, but games, visual and musical works as well.

Source code poetry is thus a form of electronic literature, but also a form of software art. Software art is an umbrella term regrouping artistic practices which engage with the computer on a somewhat direct, material

level, whether through hardware³⁴ or software³⁵. 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't 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). From this perspective, code poetry is a form of a software art where execution is required, but not sufficient to constitute a meaningful work.

The approach of code poets is therefore more specific than broad generative aesthetics: it is a matter of exploring the expressive affordances of source code, and the overlap of machine-meaning and human-meaning, acting as a vector for artistic communication. Such an overlap of meaning is indeed the 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 its inversion of the reading and executing processes. Usually, a program text is loosely assumed to be somewhat pleasurable to read, but is expected to be executable. Code poems rather assume that the program text is somewhat executable, but demand that it is pleasurable to read. 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 interpreter or compiler of the language in which they are written, but they are functional nonetheless. They are functional to the computer, in that they are composed in a legal syn-

³⁴See Alexei Shuglin's *386 DX* (1998-2013)

³⁵See Netochka Nezanova's *Nebula.M81* (1999)

tax and can be successfully parsed; but they do not need their output to do anything of immediate and measurable *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 have 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. They also make the reader reconsider the relationship to the machine, and the relationship to function. By using a machine language in the way the machine expects to receive it, it is no longer software referring to itself, exploring its own poetics and its specific meaning-making abilities. By forcing itself to be functional—that is, to produce meaningful output as the result of execution, it becomes software investigating itself, and through that, investigating the system within which it exists and acts, and the assumptions we ascribe to it. Code poems thus shed a new light on how and why source code is written, not as a functional artefact, but as a poetic one, focusing on fabrication rather than production, and expressing a subject rather than an intent (Paloque-Bergès, 2009).

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 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 overwhelming majority, English) at the highest end. This implies that they could be written and read similarly to English, including word order, pronunciation and interpretation, similar to the error-tolerance of human languages, which doesn't cause the whole communication process to fail whenever a specific word, or a word

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

Listing 20: AGC source code for the Lunar Landing Guidance Equation, 1969

order isn't understood.

Layered machine texts

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 perhaps the Apollo 11 Guidance Computer (AGC) code, written in 1969 in Assembly (Garry & Hamilton, 1969). Cultural references and jokes are peppered throughout the text as comments, asserting computer code as a means of expression beyond exclusively technical tasks³⁶, 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 20.

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, in the midst of an impersonal interaction with the machine system.

Rather than limiting their lexical field to comments, some writers decided to engage directly with machine keywords in order to compose po-

³⁶Other files include comments such as "Crank that wheel" or "Burn Baby Burn" when triggering the ignition subroutine.

ems. One of the first instances of this human poetry composed with machine syntax are the Poèmes Algol by Noël Arnaud (Arnaud, 1968). As a member of the Oulipo movement, he sets himself the constraints of only using those reserved keywords of the ALGOL 68 programming language to extract meaning beyond their original purpose. Reading those, one is first struck by their playfulness in pronunciation, and subsequently by the unexpected linguistic associations that they suggest.

More recently, this has been illustrated in the work of MOONBIT (Mosteirin & Dobson, 2019), a series of code poems computationally extracted from the AGC's source code, with those two program texts standing almost 50 years apart. In their work, the authors want to highlight that software is not only functional, but also social, political and aesthetic; importantly, the relation between aesthetics and function is not seen as mutually exclusive, but rather as supplementary³⁷. As programmers could already express themselves in a language as rigid as Assembly, subsequent programming languages would further expand poetic possibilities.

Code poetry benefited greatly from the advent of scripting languages, such as Python, Ruby or Perl (see 2.1.3 above). As we've seen, scripting languages are readable and versatile; readable because their syntax tends to borrow from natural languages rather than invented idioms, at the expense of functionality³⁸, and versatile because they often handle some of

³⁷"The aesthetic features of computer code—often characterized by a rigidly formal, restricted syntax, and numerous paralinguistic dimensions—sometimes have a supplemental character; they appear, at times, almost ornamental in their sheer excess beyond the functional elements and programmed goals. At other times, these features are an intrinsic and necessary part of the code. We believe that these special properties of computer code make possible imaginative uses or misuses by its human programmers and that these properties and features justify our exuberant readings, misreadings, translations, and appropriations." (Mosteirin & Dobson, 2019)

³⁸For instance, C's `strtok()` separates a string of text in a list of strings along several particular delimiters, while Python's `str.split()` does the same thing with a more readable name, but with only one delimiter.

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

Listing 21: Just Another Perl Hacker, japh.pl

the more complex and subtle data and platform idiosyncracies³⁹.

The community of programmers writing in Perl, perlmonks⁴⁰ 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 21.

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 have 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:

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 impor-

³⁹Python and Perl are both dynamically typed languages, meaning that the writer does not need to bother with additional syntax and possible verbosity, but rather focus only on the most expressive tokens, all while letting the interpreter deal with the kinds of errors which would undermine the functionality requirement of code poetry in other languages.

⁴⁰See their website: <https://perlmonks.org/>, with the spiritual, devoted and communal undertones that such a name implies.

tant: 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 in 1990, is another example of the richness of the productions of this community. It is presented in 22 in its updated form by kck, making it compatible for perl 5.20 in 2017. The effort of Perl community members of updating Black Perl to more recent versions of the language is a testament to the fact that one of the intrinsic qualities of the poem is its ability to be correctly processed by the language interpreter.

The most obvious feature of this code poem is that it can be read by anyone, 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 addressee adds to the ominousness of each verse.

The object of each of these predicates presents a different kind of am-

```

#!/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

```

Listing 22: Black Perl is one of the first Perl poems, shared anonymously online. It makes creative use of Perl's flexible and high-level syntax.

biguity: earlier versions of Perl function in such a way that they ignore unknown tokens⁴¹⁴². Each of the non-reserved keywords in the poem are therefore, to the Perl interpreter, potentially inexistant, allowing for a large latitude of creative freedom from the writer's part. Such a feature allows for a tension between the strict, untouchable meaning of Perl's reserved keywords, and the almost infinite combination of variable and procedure names and regular expressions. This tension nonetheless happens within a certain rhythm, resulting from the programming syntax: `kill them, dump qualms, shift moralities`, here alternating the computer's lexicon and the poet's, both distinct and nonetheless intertwined to create a *Gestalt*, a whole which is more than the sum of its parts.

A clever use of Perl's handling of undefined variables and execution order allows the writer to use keywords for their human semantics, while subverting their actual computer function. For instance, the `die` function should raise an exception, but wrapped within the `exit ()` and `close` keywords, the command is not interpreted and therefore never reaches the execution point, bypassing the abrupt interruption. The subversion here isn't purely semiotic, in the sense of what each individual word means, but rather in how the control flow of the program operates—technical skill is in this case required for artistic skill to be displayed.

Finally, the use of the BEFOREHAND: and AFTERWARDS: words mimick computing concepts which do not actually exist in Perl's implementation: the pre-processor and post-processor directives. Present in languages such as C, these specify code which is to be executed respectively before and after the main routine. In this poem, though, these patterns are co-opted to reminisce the reader of the prologue and epilogue sometimes present in literary texts. Again, these seem to be both valid in computer and human terms, and yet seem to come from different realms.

⁴¹e.g. undefined variables do not cause a core dump.

⁴²Which results in the poem having to be updated/ported, in this case by someone else than the original writer

This instance of Perl poetry highlights a couple of concepts that are particularly present in code poetry. While it has technical knowledge of the language in common with obfuscation, it departs from obfuscated works, which operate through syntax compression, by harnessing the expressive power of semiotic ambiguity, giving new meaning to reserved keywords. Such an ambiguity is furthermore bi-directional: the computing keywords become imbued with natural language significance, bringing the lexicon of the machine into the realm of the poetic, while the human-defined variable and procedure names, and of the regular expressions, are chosen as to appear in line with the rhythm and structure of the language. Such a work highlights the co-existence of human and machine meaning inherent to any program text⁴³.

Following in the footsteps of the *perlmonks*, additional communities around code poetry have formed, whether in university settings, such as Stanford's Code Poetry Slam, which ran between 2014 and 2016 (Kagen & Werner, 2016), or as independent initiatives, like the Source Code Poetry event, which runs annual contests (Unknown, 2017). The simple constraint and low barrier to entry also results in collective writings where programmers engage in playful writing, such as in the #SongsOfCode trend on a micro-blogging website where the challenge. In 23, we can see a simple example of translation from the problem of popular pop songs into machine language. The tension between the familiarity of the song and the estrangeness of the Java syntax is a kind of puzzle that is also reminiscent of hackers, further establishing cognitive complexity as a factor in the aesthetic judgment of source code poetry.

We saw in 2.1.1 that the transition of programming from an annex practice to a full-fledged discipline and profession resulted in source code being recognized as a text in its own, to which engineering and artistic attention should be paid. No longer a transitional state from formula to binary, it becomes a semantic material, whose layout, organization and syntax are

⁴³Except perhaps those which deal exclusively with scientific and mathematical concepts

```

/** Nothing compares 2 U */

public class U{
    public bool Equals(object obj){
        return false;
    }
}

```

Listing 23: #SongsInCode is an example of functional source code poetry written to represent the traditionally non-functional domain of pop songs.

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, existing parallel to the need for a program to be functional, and echoing the practice of Guillaume Apollinaire's *calligrammes*.

There, the physical layout of the program text comes to the forefront, along with its executed representation. Written by Kerr and Holden in 2014, *water.c* is a poem written in C which illustrate both of these components. In 24, we can see that the way the whitespace is controlled in the source code evokes a visual representation of water as three columns composed of ;, { and } characters, computer-understood punctuation which nonetheless holds only a tiny semantic load as block and statement delimiters.

Once compiled and executed, *water.c* gains an additional quality: its output represents moving droplets running across the screen, with a particular frame shown in 25. We see that code poetry, like other forms of writing program texts, differ from other means of expression in their dual representation, as source and software, static and dynamic.

Code poetry values code which, while being functional, expresses more than what it does, by allowing for *Sprachspiele*, languages games where pronunciation, 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. A subtle interplay of human meaning and machine meaning, layout and execution allows for a complex po-

Listing 24: water.c has a very deliberate layout and syntax, reminiscing of *calligrammes* (Holden & Kerr, 2016).

Listing 25: The output of 24 consists in ASCII representation of water droplets, bearing a family resemblance to BASIC one liners, and suggesting a complementary representation of water.

etic emergence.

From engineers to poets, this section has shown how the set of individuals who write and read code is heterogeneous, varying in practices, problems and approaches. While none of these communities of practice are mutually exclusive—a software developer by day can hack on the weekend and participate in code poetry events—, they do help us grasp how source code's manifestations in program texts and its evaluation by programmers can be multifaceted. 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 playful idiosyncracy for the purpose of immediate, functional efficiency, with a practical engagement with the tools of their trade. On the other hand, scientific programming favors ease of use and reproducibility, along with a certain quest to represent the elegant concepts of computer science, while code poets explore the semantic tension between a human interpretation and the machine interpretation of a given source code, via syntactic games, graphical layouts and interplay between the static and executed versions of software.

Still, there are strands of similarity within this 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 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? To start this investigation, we first analyze programmers' discourses in 2.2 in order to identify concrete categories of formal properties which might enable a source code to be positively valued for its appearance, before we ex-

amine the aesthetic domains code practitioners refer to when discussing beautiful code in 2.3 to further qualifies these properties.

2.2 Ideals of beauty

Following our overview of the varieties of practices and program texts amongst those who read and write source code, we now 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 in a dual approach. 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. To do so, we will introduce a discourse analysis framework for the empirical study of the corpus, followed by an examination of the discourses that programmers deploy when it comes to explicit their aesthetic preferences of source code. What we will see is that a set of aesthetic values and a set of aesthetic manifestations prove to be recurringly consistent; conversely, the aesthetic domains that are mobilized to justify these values are clearly distinct.

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 (Mullet, 2018), 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 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: expliciting what writing good code entails. 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 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). As the macro structure represents a certain kind of world situation, we will connect these to specific aesthetic fields, considering that the world of the aesthetics of source code is pragmatically connected, by the programmers and via their discourses, to adjacent world of the aesthetics of architecture, literature and mathematics.

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 such as metaphorical devices may attract attention to important concepts, provide more cues for local and global coherence, suggest plausible interpretations (e.g., a praise versus a critique), and will in general assign more structure to elements of the semantic representation, so that [meaning] retrieval is easier (Dijk & Kintsch, 1983). As we will see, a reference to code as a spaghetti is not made as connote a nutritional value, but rather convoluted spatial

properties.

Following this idea, we will proceed by examining discursive markers to deduce overarching concepts at the semantic level. Among those discursive 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 thematic entities which underline the discourse at hand. We will also not be limited to discourses in natural language, but also include source code examples presented by programmers as components of their argumentation.

Finally, our interpretation of the macrostructures described by Kintsch and Van Dijk will be complemented by the work done by Lakoff and Johnson on a theory of conceptual metaphors. They argue that the metaphor maps a source domain, made up of cognitive structures, to a target domain and, 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 (Lakoff & Johnson, 1980; Lakoff, 1980). Metaphors' essential dependence on these pre-existing cognitive structures, which we associate with familiar concepts and properties, give them an explanatory quality when it comes to qualify foreign domains.

In particular, these sources are defined enough to not be mistaken for something else, but broad enough to allow for multiple variants of itself to be applied to various targets, providing both diversity and reliability in our inquiry.

As we will see below, their approach allows us to focus not just on textual objects, but on the vast range of linguistic devices used to make sense in computing-related environments. Given that the source of the metaphor should be grounded, with as little invariability 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* mentioned in 3.3.1, we will explore Lakoff's understanding of the specifically poetic metaphor in 4.2.1 when it comes to qualifying the aesthetics of source code. We will pay particular attention to what programmers are saying about beautiful (or ugly) 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 chapter, 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 discussions⁴⁴. These constitute our primary sources insofar as they are written by practitioners on the topic of good and beautiful code. 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. We consider that authoritative sources can be both canonical textbooks or widely-read blog posts from well-known skilled practitioners, but also include more casual forum exchanges in order to support the empirical dimension of our research. This methodology will allow us to show that there are *specific* ways in which programmers qualify well-written code, and employing recurring references.

⁴⁴Specifically, 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.

2.2.2 Lexical Field in Programmer Discourse

There is one major study of the lexical field programmers use, done by Erik Piñeiro 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, we also build upon it in several ways. First, Piñeiro focuses on a narrower corpus, that of the Slashdot.org forums (Pineiro, 2003). 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), 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 a broader interdisciplinary field including comparative media studies and aesthetic philosophy and science and technology studies. Still, Piñeiro's work provides valuable insights in terms of identifying the manifestations and rationales for an aesthetic experience of source code. Here, we build on his works by highlighting the main adjectives in the lexical field of programmers' discourse, in and beyond software developers.

Clean

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 developed at great lengths a couple of decades later in 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 aim⁴⁵. What exactly is cleanliness, in Martin's terms, is nonetheless defined by circumlocutions; he relies on contributions from experts, again showing the relation ship between expertise and aesthetic judgment. 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: by being devoid of any extraneous syntactic and semantic symbols, it facilitates the identification of the core of the problem at hand. Cleanliness thus works as a pre-requisite for

⁴⁵*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.*

expressivity. In a clean-looking program text, the extraneous details disappear at the syntactic level, in order 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)

Here, it is about quantity rather than quality: ornaments that are positively valued in parsimony (such as comments) can prove to be detrimental when there are too many of them. 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 across programmer discourses (Programming Wisdom [@codewisdom], 2021; Jackson, 2010; 4.4.7, 2003):

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)⁴⁶

Obfuscation

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 have already seen in the works of the IOCCC

⁴⁶ *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 (Saint-Exupery, 1990)

```

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 26: home.js is an excerpt of a JavaScript program text as it is written by a human programmer, before minification.

above (see the discussion around 14). 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 26 is obfuscated through a process called *minification* into the source code in 27. The result is a shorter and lighter program text when it comes to its circulation over a network, at the expense of readability.

In most cases, this process of obfuscation has very defined, quantitative assessment criterias, such as the size of the source code file and crypto-

```

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,H as g,e as w}from"./Header.js";const
H={class:"container p-3"},N=e("h1",null,"Home",-1),k={class:"syll
abi"},x=[{"href"},B={class:"cta"},F=m({setup(S){const
s=v(""),d="http://localhost:3046",o=new Array;let
h=()=>{window.location.href="/cartridge.html"};return f(()=>{fetc
h(`${d}/syllabi/`,{method:"GET"}).then(t=>t.json()).then(t=>{Obje
ct.assign(o,JSON.parse(t)),console.log(o),o.length==0?s.value="No
syllabi :(s.value=`There are ${o.length}
syllabi.`)}.catch(t=>{console.error(t),s.value="Network error
:"})}),t,i=>(l(),n(u,null,[c(g),e("main",H,[N,e("div",k,[e("di
v",null,r(s.value),1),e("ul",null,[(l(!0),n(u,null,b(_(o),a=>(l())
,n("li",null,[e("div",null,[e("a",{href:"/syllabi/"+a.ID},r(a.tit
le,9,x)]),e("div",null,r(a.description),1))))],256))])),e("div"
,B,[e("button",{"id":"cta-upload",class:"btn btn-primary mb-4
cc-btn",onClick:i[0]||(i[0]=a=>_h()),"Upload
yours!"})]),c(y)],64))});var O=p(F,[["__file","/home/pierre/cod
e/commonsyllabi/viewer/www/src/Home.vue"]]);w(O).mount("#app");

```

Listing 27: home.js is the same program as in 26, after minification. Syntactical density is gained at the expense of clarity.

graphic complexity (Pellet-Mary, 2020). Nonetheless, obfuscation can also be valued as a positive aesthetic standard, of which the IOCCC is the best example and the most institutionalized guarantor. These kinds of obfuscations, as Mateas and Montfort analyze, involve the playful exploration of the interwindings 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 coding 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 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. Through its negative connotations, obfuscation nonetheless points at the recurring theme of ease (or difficulty) of understanding.

Simple

This balance between too much and too little is found in another dichotomy stated by programmers: between the 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).

Simplicity in source code is therefore a form of parsimony and balance⁴⁷.

This requirement of exerting balance leads us to make a difference between two kinds of simplicity: syntactical simplicity, and ontological simplicity. Syntactic simplicity measures the number and conciseness visible lexical tokens and keywords. Ontological simplicity, in turn, measures the number of kinds of entities involved in the semantics of the program text. Source code can have syntactic simplicity because it wrangles together complex concepts in limited amount of characters (see our discussion on one-liners in 2.1.2), or code is ontological simplicity, because of the minimal amount of computational concepts involved (as explained in 2.1.3). Syntactical simplicity also has a more immediate consequence on one's

⁴⁷Gibbons quotes Ralph Waldo Emerson to qualify his point: "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)

experience when reading a program text: one of the issues that programmers face is that there are just too many lines of code that one can wrap its head around, thus requiring that the content be pared down to its functional minimum (Butler, 2012).

This distinction between syntactical and ontological simplicity highlights this need for balance, along with the concrete tradeoffs between syntax and semantics that might need to be done when writing code. Source code aesthetics thus have to balance between simplicity in breadth and simplicity in depth regarding the composition of the program text, between the precision of a use-case in a problem domain and its generalization, and between its self-reliability and its leveraging of external—i.e. supposedly reliable—program texts.

In another ACM publication, Kristiina Karvonen argues for simplicity not just as a concrete design goal, as leveraged by human-computer interface designers⁴⁸, 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, we take her correlation between *significance* and *beautiful* in a very literal manner; a connection between significance and beauty hints at the semantic role of beauty, and thus of simplicity as a component of the beautiful, at the role of beauty as a means to communicate (i.e. to *signify*) ideas to an audience.

Precisely, simplicity is correlated with clarity (of meaning); if the former refers mainly to the syntactical and ontological components, it enables the non-obfuscated representation of the ideas at play in the function of a program text. An example of clarity is given in 28 by Dave Bush in a post titled *15 Ways to Write Beautiful Code*.

Here, the strive for simplicity implies removing the brackets, and flip-

⁴⁸The field of human-computer interfacing does not limit itself to graphical user interfaces; a software library can act as textual interface between a human and a machine system.

```

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

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

```

Listing 28: Example of clarity differences between two methods.

ping the boolean check in the if-statement to add an early `return` statement. 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, by trading syntactical simplicity for ontological simplicity. Bush 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 true*, 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 true* 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, at the bottom of the program text (Bush, 2015).

A final 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. He argues that programmers have a desire to increase the semantic charge (or significance, in Karvonen's terms) all the while reducing the syntactic load (or the quantity of formal tokens). Compression, as we will see in 4.3.2, implies simplicity, but also qualifies such simplicity in terms of how much is expressed by a simple statement. The more complex the problem the program intends to solve, the more important the role simplicity plays in communicating such complexity. 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)

Simplicity is found in source code when the syntax and the ontologies used are *an exact fit to the problem*: simple code is code that is neither too precise, nor too generic, displaying an understanding of and a focus on the problem domain, rather than the applied tools.

Cleverness

Conversely, the intellectual nature of a programmer's practice often involves technical tricks. Even though programming is both a personal and collective activity, there is a tendency of programmers to rely on convoluted, *ad hoc* solutions which happen to be quick fixes to a given problem, but which can also be difficult to generalize or grasp without external help. Such an external help often takes the form explanation, and is not often positively valued, as pointed out online by Mason Wheeler:

When it requires a lot of explanation like that, it's not "beautiful code," but "a clever hack." (Overflow, 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?" (Overflow, 2013), not only highlights the ideal for a program text to be self-explanatory, but also points at a quality departing from simplicity—cleverness.

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 reproduced in 29 as an example of clever, and therefore bad, code:

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

Listing 29: unique.py: A function to check for the uniqueness of array elements, using a very specific feature of the Python syntax, and as such an example of clever code.

From the name of the function, `is_unique()`, one can deduce that what the program text does is returning whether all elements of a list are unique. However, to understand the particular way in which this is done, the writer requires knowledge of how the `set()` function in Python behaves. A programmer without familiarity with Python would be unable to do so without consulting the Python documentation, or through external explanation.

Hillel elaborates on the difference between "bad" clever code⁴⁹, which is essentially 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.

In other contexts, cleverness can be valued positively. Hacker practices in particular tend to put more emphasis on the technical solution than on the problem domain, as we have saw in 2.1.2. A salient example was the 1994 `smr.c` entry to the IOCCC, which aimed at being the smallest self-reproducing program (Kanakarakis, 2022). An exact reproduction of the source code can be found in 30

⁴⁹See, 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 30: smr.c

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, 2022), resulting in a particular confusion to the reader (and jury). 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 socio-technical environment.

Elegance

Programmers hold the idea of reaching a aesthetic quality through the reduction of complex syntactical and ontological constructs, without minimizing expressivity. 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*. Such an ideal is clearly rooted in the definition of elegance given by the *Jargon File*, also known as the hacker's dictionary:

elegant: adj.

[common; from mathematical usage] Combining simplicity, power, and a certain ineffable grace of design. Higher praise than 'clever', 'winning', or even cuspy.

*The French aviator, adventurer, and author Antoine de Saint-Exupéry, probably best known for his classic children's book *The Little Prince*, was also an aircraft designer. He gave us perhaps the best definition of engineering elegance when he said "A de-*

signer knows he has achieved perfection not when there is nothing left to add, but when there is nothing left to take away." (4.4.7, 2003)

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 load 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 5.1.1.

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. 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. Ideally, elegant code communicates the problem it solves and the machinery of its solution, all through a single lens.

This aspect of implying underlying principles is also present in Bruce McLennan's discussion of the concept. He also adds to this perspective a certain subjective feeling. 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. Interestingly, he approaches elegance through the dual lens of structural and software engineering, this indicates that he also considers elegance as a more profound concept which can manifest itself across disciplines, connecting ways of making, and ways of thinking (McLennan, 1997).

bothOn 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. (Overflow, 2013)

Connected to simplicity by way 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 have seen so

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

Listing 31: factorial.c: The use of recursion, rather than iteration, in the computation of a factorial is particularly praised by programmers.

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 31.

Recursion, or the technique of defining something in terms of itself, is a very positively valued feature of programming (Abelson et al., 1979), which we have seen an example of in 17. 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 32)

Going back to the personal factor in perceiving elegance, 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, a duality whose implications are developed in 3.2.

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 have 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 comple-

```

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 32: The comparison two functions, one using recursion, the other one using iteration, intends to show the computational superiority of recursion. (amit, 2012).

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

Listing 33: Choose variable names that masquerade as mathematical operators

mentary 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 , written by one for another.

Smells

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 negative qualifiers. We have already touched upon terms 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 examination of negative qualifiers will enrich of understanding of what constitutes good code; programmers have another way to refer to code that does not meet aesthetic criteria, by referring to material properties.

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 33 or semantic interactions in 34. 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

```

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 34: Code That Masquerades As Comments and Vice Versa

is doing. The opposite amounts to playing misleading tricks.

For instance, *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 technically 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 is declinated in a similar way 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 explicit bugs. The name code smell evokes the fact that their recognition happens through intuition and experience of the programmer reading the code, invisible yet present, rather than through careful empirical analysis⁵⁰. This points to a practice-based skill system to evaluate the quality of source code, rather than to an evidence-based one, itself circling back to the qualifications of elegance discussed above, evaluated both as quantitative metric and as qualitative one.

In conclusion, this section has clarified some of the key 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: excellence in instrumental action forms the core of writing source code, but can also be declinated along different contexts of reading and writing. 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 smelly).

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, with aesthetics helping alleviate such a burden. Beautiful code renders accessible the ideas

⁵⁰It 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)

embedded in it, and the world in which the code aims to translate and operate on. 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 can see in the breadth of responses in a programmer's question of "How can you explain "beautiful code" to a non-programmer?" (Overflow, 2013) that programmers also rely multiple aesthetic domains to which they refer: from engineering and literature to architecture and mathematics. As such, they deploy metaphors for what beautiful code is. Moving from a syntactical level to a thematical level, to refer to Kintsch and Van Dijk's framework of discourse analysis, we now turn to an investigation of each of these domains, and what they tell us about source code.

2.3 Aesthetic domains

The 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 complicated to define: as we have seen in the previous section's discussion of forum exchanges, beauty in source code is not explicated in and of itself.

The next step we propose is to inquire into the specific domains that programmers use to illustrate the qualities of source code; we will examine in which capacity these are being summoned in relation to code, and how they help us further delineate the aesthetic qualities of source code. The assumption here is that a medium—such as source code—is a means

of expression, and different mediums can support different qualities of expression; additionally, a comparative analysis can be productive as it reveals the overlaps between these mediums. Since there seems to be some specific ways in which code can be considered beautiful, these adjacent 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 music(McLean, 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, film, or dance, 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, they both involve syntax and semantics interplay in order to convey meaning to a reader. *Code as literature*, then, focuses on this similarity 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 nonethe-

less 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 Āryabhaṭa (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 shared by the writers of source code. 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 "*connoisseurship*", 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 (Hayles, 2004).

Specifically, Hayles looks at how both literature and code can be expressive in both a syntagmatic and 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, the meaning across synonyms in a (program) text is always potential in literature, but always present in code, thus highlighting different levels of interpretation (Hayles, 2004). 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 can thus be 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, 2008) (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 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 literature⁵¹.

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 compile a single source into either a typeset documentation of the program, using the TeX engine, 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. As this approach fits within Knuth's interest in typesetting and workflows of scientific pub-

⁵¹For instance, he refers in the rest of the article as "constructing" programs, rather than "writing" them.

lications, it first locates the relationship between literature and programming beyond this formal level.

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 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 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 co-exist⁵². We also note here the phrase "*psychologically correct order*", highlighting the psychological dimension involved in a programmer's activity, further developed in 3.2.3. Indeed, code is never read linearly, as most codebases might have an *entrypoint* but no *introduction*⁵³.

Moving away from this hybrid approach involving both natural and machine texts, Yukihiro Matsumoto, the creator of the Ruby programming language, develops his notion of *code as an essay* in his contribution to the edited volume *Beautiful Code* (Oram & Wilson, 2007). While he does not 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

⁵²See JavaDocs, Go docs, Jupyter Notebooks

⁵³"Having conducted interviews with several leadig programmers, Peter Seibel comes to the conclusion: "We don't read code, we decode it. We examine it. A piece of code is not literature; it is a specimen." (Seibel, 2014)

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. (Matsumoto, 2007)

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) (Barthes, 1977). 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. It is these appropriations, or uses, that give a writerly text its value.

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 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 5.1.3.

These two examples argue that source code can be considered a text which needs to accommodate 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, providing cognitive cues for the programmer who will re-use or modify the program.

There is, however, a remnant of readerly texts in the literary conception of source code. 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 on Code Art*⁵⁴, 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-

⁵⁴A direct reference to Sol Lewitt's *Sentences on Conceptual Art*.

generated. We bring to code our excesses of language, and an ambiguity of semantics, as discerned by the human reader. (Temkin, 2017)

The artists whose main medium is source code explore the possibilities of meaning-making through mechanisms usually associated with poetry, in both its spoken, written and executed form⁵⁵. Code poetry is a particular kind of writing source code, one which is focused on the evocative possibilities of machine languages, and the generative interpretation of its human readers, and away from an explicitly productive function. 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 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 #!.

⁵⁵Evidently, code works like poetry in that it plays with structures of language itself, as well as our corresponding perceptions. (Cox, 2011)

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

Listing 35: All The Names of God, Nick Montfort, 2010, source

In the former's foreword, Jamie Allen develops 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 considered an essential device in writing code poetry, and exploring concepts that are not easily grasped in natural languages—e.g. callbacks, asynchronous promises or destructuring assignments. Additionally, the contrast between the source representation of the poem and its execution can add to the poetic tension, as we saw in 24, and here in Nick Montfort's *All The Names of God* (2010) (source in 35, and output in 36).

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 expressive 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). In between an infinite output and a one-line hack, *All The Names of God* is in the form monos-

```

_atk_atl_atm_atn_ato_atp_atq_atr_ats_att_atu_atv_atw_atx_aty_atz_aу_
→ а_aub_auc_aud_aue_auf_aug_auh_aui_auj_auk_aul_aum_aun_auo_aup_а_
→ uq_aur_aus_aut_auu_auv_auw_aux_auy_auz_ava_avb_avc_avd_ave_avf_ъ_
→ avg_avh_avi_avj_avk_avl_avm_avn_avo_avp_avq_avr_avs_avt_avu_avv_ъ_
→ _avw_avx_avy_avz_awा_awb_awc_awd_awe_awf_awg_awh_awi_awj_awk_aw_
→ t_awm_awн_awo_awp_awq_awr_awс_awt_awu_awv_aww_awx_awy_awz_axa_a_
→ xb_axc_axd_axe_axf_axg_axh_axi_axj_axk_axl_axm_axn_axo_axp_axq_ъ_
→ axr_axs_axt_axu_axv_axw_axx_axу_axz_aya_ayb_ayc_ayd_aye_ayf_ayg_ъ_
→ _ayh_ayi_ayj_ayk_ayl_aym_ayn_ayo_ayp_ayq_ayr_ays_ayt_ayu_ayv_ay_
→ w_ayx_ayy_ayz_аза_azb_azc_azd_aze_azf_azg_azh_azи_azj_azk_azl_а_
→ zm_azn_azо_azp_azq_azr_azs_azt_azу_azv_azw_azx_azу_azz_baa_bab_ъ_
→ bac_bad_bae_baf_bag_bah_bai_baj_bak_bal_bam_ban_bao_bap_baq_bar_ъ_
→ _bas_bat_bau_bav_baw_bax_baz_bba_bbb_bbс_bbd_bbe_bbf_bbг_bbв_bb_ъ_
→ h_bbi_bbj_bbк_bbл_bbм_bbн_bbо_bbр_bbq_bbг_bbт_bbт_bbу_bbв_bb_ъ_
→ bx_bby_bbз_bbа_bbс_bbд_bbс_bbд_bbс_bbв_bbс_bbд_bbс_bbв_bb_ъ_
→ bcn_bco_bcp_bcq_bcr_bcs_bct_bcu_bcv_bcw_bcх_bcy_bcз_bda_bdb_bdc_ъ_
→ _bdd_bde_bdf_bdg_bdг_bdi_bdj_bdk_bdl_bdm_bdn_bdo_bdp_bdq_bdr_bd_
→ s_bdt_bdu_bdv_bdw_bdx_bdy_bdz_beа_beb_bec_bed_bee_bef_beg_beh_b_
→ ei_bej_bek_bel_bem_ben_beо_bep_beq_бер_bes_bet_beу_bev_bew_bex_ъ_
→ bey_bez_bfa_bfb_bfc_bfd_bfe_bff_bfg_bfh_bfi_bfj_bfk_bfl_bfm_bfn_ъ_
→ _bfo_bfp_bfq_bfr_bfs_bft_bfu_bfv_bfw_bfx_bfy_bfz_bga_bgb_bgc_bg_
→ d_bge_bgf_bgg_bgh_bgj_bgj_bgk_bgл_bgm_bgн_bgo_bgp_bgq_bgr_bgs_b_
→ gt_bgu_bgv_bgw_bgx_bgy_bgz_bha_bbh_bhc_bhd_bhe_bbh_bbh_bbh_ъ_
→ bhj_bhк_bhл_bhм_bhн_bhо_bhр_bhq_bhр_bhс_bhт_bhу_bhв_bhw_bhх_bhу_
→ _bhз_bia_bib_bic_bid_bie_bif_big_bih_bii_bij_bik_bil_bim_bin_bи_
→ o_bip_biq_bir_bis_bit_biu_biv_bix_biy_biz_bja_bjb_bjc_bjd_b_
→ je_bjf_bjg_bjh_bji_bjj_bjk_bjl_bjm_bjn_bjo_bjp_

```

Listing 36: All The Names of God, Nick Montfort, 2010, Selected output

tiche, a natural language poem composed of a one-line stanza, where the quality aesthetic quality of minimalism is correlated with its expressive power.

Not only is there an aesthetic of minimalism present in the source, the output also represents the *depth* (in Hayles's sense) of the medium of writing. In this case, source code also supports academic literary analysis, thus reinforcing a literary conception of source code aesthetics.

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. One perspective is focused on its need to communicate explicit concepts related to its function (Knuth, Matsumoto, Brock), while a complementary perspective embraces 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).

This tension, between functional efficiency of the text, and dramatic expressiveness of the poem, suggests a parallel with scientific practices, this is something that Andrei Ershov points to in his 1972 address to the Joint Computer Conference:

"A professional aesthetic influences and is influenced by the ethical code of a profession, by the technical subject matter of the profession, and by the profession's juridical status. [...] The creative nature of programming does not require special proof. Indeed, I may assert, programming goes a little further than most other professions, and comes close to mathematics and creative writing." (Ershov, 1972)

2.3.2 Scientific beauty

Rooted in computer science's thought and practice, the aesthetic experiences of source code are also related to the scientific domain. Specifically, it seems to exist in two distinct ways: whether code is beautiful in a similar way that mathematics is, or whether code is beautiful according to 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.

Among the few discourses of a direct relation between code and beauty from a mathematical perspective, we can see 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).

This parallel between source code and mathematics becomes clearer when looking at the kinds of aesthetic effects which mathematics possess. 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 imple-

mentation 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 elegance 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 cumbersoness) (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 by the reader. This, 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 mathe-

```

template <std::size_t V>
auto floyd_marshall(std::array<std::array<int, V>, V> const &graph) {
    auto dist = graph;

    for(auto k: std::views::iota(0u, V))
        for(auto i: std::views::iota(0u, V))
            for(auto j: std::views::iota(0u, V))
                if(dist[i][k] + dist[k][j] < dist[i][j])
                    dist[i][j] = dist[i][k] + dist[k][j];

    return dist;
}

```

Listing 37: Implementation of the Floyd-Warshall algorithm, showing an elegant implementation of a complex theory.

matics 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.

One can find such connections between mathematical and source code elegance in their conciseness to express established, complex ideas. For instance, the implementation of the Floyd-Warshall algorithm reproduced in 37 is considered by Sy Brand as eliciting an aesthetic experience (CPPP Conference, 2022).

Brand's discussion of his aesthetic experience highlights another aspect of source code beauty: intellectual engagement. In order to appreciate the aesthetics of a program text, one needs to take an active stance and understand what it is that the code (in the case of 37 does, the function) is trying to do. Once that is understood, one can then appreciate the way in which the algorithm is implemented—that is, its aesthetics.

Engineering

As we have seen in our discussion of the relationship between computer science and programming as a relationship between the abstract and the concrete, one can see in these two activities a parallel in mathematics

and engineering, considered as both scientific endeavours. 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 problem⁵⁶. Mathematics, from this perspective, can be considered as one of the languages of engineering, among sketches, diagrams, techniques, tools, etc.

Nonetheless, 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 aesthetics in engineering also play a cognitive role:

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 are not always enough to understand what is going on in a given program text. 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, albeit in different ways. While the latter can rely on succinct formal propositions and representations to achieve this purpose, engineering composes too many moving parts of different nature.

⁵⁶Indeed, software development is also referred to as software engineering (Bourque & Fairley, 2014); we chose to refer to the former due to its referencing to a broader set of practitioners.

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. 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 for mathematics, in which larger implications are gained from a particular way a proof of a theorem is presented, or as an encompassing *gestalt* impression in engineering, in which a program that looks correct, would most likely be correct.

Two concepts touched upon by both approaches are that of structure and know-how. While mathematics deal with formal structures to represent and frame the complexity of the world, engineering deals with concrete structures offered as solutions to a specific problem. In both domains, there is also a reference to a certain sense of intuition, which enables cognitive discovery of a functional artefact (whether an abstract theorem or a concrete construction), something we also find when exploring parallels with architecture

2.3.3 Architectural beauty

Beyond a more official understanding of software architecture (see 2.1.1), architecture is used extensively as a metaphor for code. In this section, we will look at architecture from two complementary perspectives: as a top-down approach, and as a bottom-up practice. This will allow us to touch on notions of structure, form and function, and provides us with another perspective which will bring into light the idea of craft.

Formal organization

Software architecture emerged as a consequence of the structured revolution (Dijkstra, 1972), which was concerned more with the higher-level organization of code in order to ensure the quality of the software produced. Such an assurance was suggested by Dijkstra 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, complementing syntax, has therefore been an essential component of the intelligibility of software since the 1970s. It is only in the late 1990s that software architecture has been recognized as a distinct discipline, and completely separated from the actual act of programming.

[...] 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 mark the beginning of a trend of so-called architectural practices within the field of software development. These two fields overlap on the topic of structure. Through rigorous, high-level formal organization, 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.

As an example, the Linux Kernel's architecture can be considered one of the reasons why the project became so popular once integrated into the GNU ecosystem. Along with its distribution license, two of its defining features are speed and portability. While speed can be attributed to its use of C code, also responsible to some extent for its portability, the architecture of the kernel is separated in multiple components which make its extension simple. On one side is the monolithic architecture of the kernel, in which process and memory management, virtual file systems, input/output schedulers, device drivers and network interfaces are all lumped together in kernel space. This tight integration would result in a high-barrier to entry for potential contributors: in such a monolithic system, it is hard to know how a change to a part of the system would affect other parts. However, this architecture also allows for dynamically loadable kernel modules, software components which can be added and removed to the operating system without interference with the core features. This provides a quality of extendability which further contributes to the success of the ecosystem of the Linux ecosystem: there is a reliable core, but also room for extension.

An architecture, such as that of the Linux kernel, thus provides significant *semantic* content about the kinds of properties that developers should be concerned about and the expected paths of evolution of the overall system, as well as its subparts. The blueprint of the software is made clear enough that it is simple for programmers to find a correct way to contribute to it. Other architectures include, for instance, the client-server architecture (with the peer-to-peer architecture as an alternative), the model-view-

controller architecture (and its presentation-abstraction-control counterpart)⁵⁷. In all of those cases, a familiar organization of a program texts files and delimitations of its functions lowers the barrier to entry for a programmer, and in this sense contributing to making the program texts writerly texts.

Eric Raymond develops this praise of the Linux kernel in his book *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.

Concepts such as modularity, spatial organization or inter-dependence, it turns out, could be applied to both fields. There are only few explicit references to beauty in software architecture design; instead, desirable properties are those of performance, security, availability, functionality, usabil-

⁵⁷One can even find their source in chip design, with Friedrich Kittler famously claiming that the last people who ever truly wrote anything were the Intel engineers laying out the plan of the 8086 chip (which would engender the whole family of x86-based devices) (Kittler, 1997). In this case, this instance is one of the few which relates software architecture to its physical counterpart, albeit in a very technical sense of plans and diagrams.

ity, modifiability, portability, reusability, integrability and testability. Perhaps this is due to the fact that the traditional understanding of beauty in terms of external manifestation—decoration—isn't here the main point of software architecture, but rather a functional perception of it.

Overall, this functional conception of architecture can also be found in the trade literature. 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 (整理) - naming and sorting all components used
- seiton (整頓) - placing things where they belong
- seisō (清掃) - cleanliness
- seiketsu (清潔) - standardization and consistency in use
- shitsuke (躰) - self-discipline

This confirms the 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 the case of programming, with correct naming, correct placement, correct appearance and correct use.

This does not mean that the *a priori* distant approach to software architecture, one which excludes any concrete writing of source code, negates any sort of personality. Style is indeed present in software architecture. In this context, an architectural style typically specifies a design vocabulary, constraints on how that vocabulary is used, and semantic assumptions about that vocabulary. For example, a pipe-and-filter style might specify a vocabulary in which the processing components are data transformers (filters), and the interactions are via order-preserving streams (pipes). When it comes down to programming such an architectural style, pipes and filters

do have a very real existence in the lines of source code. These concepts are inscribed as the `|` character for pipes, or the `.filter()` method on the JavaScript array type, which itself has different ways of being written (e.g. with an anonymous callback function, or an externally defined function). By virtue of there being different ways being written, one can always argue for whether or not one is better than the other, ultimately resulting in better, clearer program texts.

More specifically, the aesthetic manifestations in the form of source code enter in a dialogue with software architecture. If a good system architecture should first and foremost exhibit conceptual integrity (Spinellis & Gousios, 2009), one can extend this integrity to its source code manifestation. A message-passing architecture with a series of global variables at the top of each file, or an HTTP server which also subscribes to event channels, would look ugly to most, since they betray their original organizational concept. These concrete manifestations of a *local texture of incoherence*, to paraphrase Beardsley, might be more akin to a *code smell*, a hint that something in the program might be deeply wrong.

Among architectural styles, it seems that brutalism is the one that tends to be equated the most with styles of programming. Simon Yuill, in the volume edited by Olga Goriunova and Alexei Shulgin, develop a parallel between code art and this style of architecture. 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.

Good source code, from a software architecture perspective, is code which is clearly organized, respecting a formalized blueprint, but does not need to exclude the reality of written lines of source code. A combination of these properties, and acknowledgment of the medium used, can then support an aesthetic experience. 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)

Both in Yuill and in Hill, we find an interesting parallel in the mention of materiality. Source code might at first seem to be immaterial, consisting of layered representations of electrical current, there is nonetheless a certain kind of tangibility which can be pointed to. Lines being re-arranged, symmetrical or out of alignment, blocks being cut and pasted, these operations all hint at a certain material engagement with the program text, rather than with its abstract model of software architecture. Considering architecture as a bottom-up practice of constructing spaces, one can turn to programmers' discourses on craft to support this material conception.

Crafting software

Considering architecture as a strictly organizational practice does not show the whole picture, as there is another side to architecture, concerned with details rather than with plans, feeling rather than rationalizing.

In their introduction to the field of software architecture, Shaw and Garlan summon the need to formalise the practice as the practice moves from craft to engineering (Shaw & Garlan, 1996). Originally, the reality of carefully crafted, individualized code and unconstrained approaches to writing code⁵⁸ was looked down upon by Dijkstra, Knuth and other early software practitioners, for its idiosyncracy and lack of rigor.

However, the conception of programming as a craft has become more and more popular amongst source code writers and readers (Spolsky, 2003; Seibel, 2009). For instance, Paul Graham, LISP programmer, co-founder of the Y Combinator startup accelerator and widely-read blogger, 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. Craftsmanship as such is an ever-fleeting phenomenon, a practice rather than a theory, in the vein of Michel De Certeau's tactics, bottom-up actions designed and implemented by the users of a situation, product or technology as opposed to strategies, in which ways of doing are prescribed in a top-down fashion (de Certeau et al., 1990).

Explicit knowledge, in programming as in most disciplines, is carried through books, academic programs and, more recently, web-based content

⁵⁸See The Story of Mel, A Real Programmer, a folktale of early programmers where hand-made code is both incredibly fast and incredibly hard to understand (Nather, 1983).

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, the road to good code is unclear, particularly when communicated in such a highly-formal language as diagramming (Munroe, 2011). 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.

The acquisition of such implicit knowledge in programming is reinterpreted 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. 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 (Depaz, 2021).

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.

So while programmers are used 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 examples of good work⁵⁹.

Considering programming a craft therefore raises questions of practice and knowledge, but also of standards of quality. In terms of aesthetic experience, it also hints at the role that style, ornament and function play in the value assessment of a well-crafted program text, just as in a well-crafted program text. These themes will act as a recurring thread throughout this study. Specifically, we will discuss the role of tacit knowledge in the programming practice in 3.1.2, and the role of tools in 3.3.2; in terms of aesthetics, the place of style between individual and collective will be analyzed in 5.1.3 before developing a further approach code's material aesthetics as refined knowledge in 4.3.3

Ultimately, architecture, when referenced by software, includes at least two distinct approaches: a top-down, formal design, and a bottom-up, materialist approach, reflected in how software also refers to architecture: as abstract planning or as hands-on construction, both holding different, but overlapping aesthetic standards. One the one side, we have cleanliness, orderliness and appropriateness, following interpersonal conventions; on the other side, we have a highly individual and informal practice of making which subsists along its explicit counterpart.

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

⁵⁹Coming back to the relationship between architecture and software, Christopher Alexander asks, in the preface of Richard P. Gabriel's *Patterns of Software*: "For a programmer, what is a comparable goal? What is the Chartres of programming? What task is at a high enough level to inspire people writing programs, to reach for the stars?" (Gabriel, 1998).

will be more thoroughly explored in 4.3. 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.

When talking about the aesthetics of source code, programmers tend to refer to three main, different domains. Each of these both reveal and obscure certain aspects of what they value in the reading and writing of program texts.

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 syntax used to write programming, its relationship to natural language and its potential for expressivity.

Considering the formal nature of source code, scientific metaphors 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 practice and in outlook, touch upon

another subject: the place of formal and informal knowledges in the construction, maintenance and transmission of those (software) structures.

In this first chapter, we laid out the ground work for our discussion of source code aesthetics. This groundwork is composed of several aspects. First, we have established the diversity of who writes code: far from a homogeneous crowd which would reflect an abstraction conception of "code", code writers include individuals who might share the practices of engineers, hackers, scientists or poets. While these categories do not have rigid boundaries and 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. 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 built on Erik Piñeiro's work to complete a survey of the lexical fields that programmers use when they describe or refer to beautiful code. In so doing, we have highlighted certain desirable 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 a certain aesthetic inclination 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 "big ball of mud", a "pile of spaghetti" or full of "smelly corners", ugly code is something where its appearance prevents the reader or writer to grasp its true purpose—what it actually does.

While those terms are being recurrently used to qualify aesthetically

pleasing code, our survey has also pointed to specific domains which programmers use as metaphors to communicate the nature of their aesthetic appreciation: by referring to science, literature and architecture. Each of these metaphors, sometimes simultaneously, select specific parts from their source domain in order to adapt to inform one's appreciation of good source code. Literature brings linguistics, but not narrative; science brings formalism and engineering, but not style nor individuality; architecture brings structure and craft, but not building codes nor end-usage. These domains are thus better understood as the different parts of a Venn diagram, as practitioners 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 how craft practices inform relations between skill, knowledge, function, space and beauty.

The overlap of these different domains has to do, it turns out, with cognitive clarity. Whether wrangling with the linguistic tokens in literary exercises, as Geoff Cox puts it:

It may be hard to understand someone else's code but the computer is, after all, multi-lingual. In this sense, understanding someone else's code is very much like listening to poetry in a foreign language - the appreciation goes beyond a mere understanding of the syntax or form of the language used, and as such translation is infamously problematic. Form and function should not be falsely separated. (Cox, 2011)

One function of aesthetics might thus be in structuring various pieces of code such that their organization is robust and communicated to others such that it allows for future maintenance and expansion. Another might be writing lines of code in a certain way in order to hint at some larger concepts and ideas beyond their immediate execution result such as in hacking or code poetry. In any case, 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.

Before we investigate precisely how aesthetics enable the understanding of computer programs, we will first explicit what makes software a cognitively complex object. The next chapter first highlights the status of software as an *abstract artifact*, before investigating the means that programmers use to understand the computational phenomena that happen at their fingertips.

Chapter 3

Understanding source code

Aesthetics in source code are thus primarily related to understanding. In the previous chapter, we have 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 computational object is a key determinant in the value judgment as applied to source code. In this chapter, we focus on the reason for which software involves such a cognitive load, before surveying the means—both linguistic and mechanistic—that programmers deploy in order to relieve such a load.

This requirement for understanding, whether in a serious, playful or poetic manner, 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 recipes¹, 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 always going to be of less value than correct software (Hill, 2016), regardless of how aesthetically pleasing the

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

source is. Any value judgment regarding the aesthetics of the source code would be subject to whether or not the software functions correctly, and such judgment is rendered moot if that software does not work.

The assessment of whether a piece of software functions correctly can be broken down in several sub-parts: knowing what the software effectively does, what it is supposed to do, being able to tell whether these two things are aligned, and understanding how it does it. 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). Due to its writerly nature, one must also understand how a program text does it, in order to modify it.

This chapter examines what goes into understanding source code: given a certain nature of knowledge acquisition, we look at some of the features of computers that make them hard to grasp, and the kind of techniques are deployed in order to address these hurdles. 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 in response to facilitate that task. Ultimately, we show that, given our definition of understanding, the complex nature of software objects and the diverse techniques programmers use to grasp these objects, aesthetics of source code also hold a significant place in this understanding process, a position we develop in 5.

The first part will lay out our definition of understanding, presenting it as a dual phenomenon, between formalism and contextualism. Starting with 20th century epistemology, we will see that theoretical computer science research has favored a dominantly rational, cognitivist perspective on the nature of understanding, eschewing another mode of understanding suggested by craft practices.

Having highlighted this tension, we then turn to how understanding

the phenomenon of computation specifically, starting from an ontological level. The ontological approach will show some of the features of software give it the status of an *abstract artifact* (Irmak, 2012), and thus highlighting in which ways is software a complex object to grasp. We then complement this ontological perspective by a more practical, psychological approach. This will show how such a comprehension takes place for situated programmers, at different skill levels, anticipating how aesthetics can fit in this model.

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 then see to what extent they differ from the metaphors used by programmers in order to understand the systems they build and work with. In the end, particular attention will be paid to their extended cognition the technical apparatuses used in the development and inspection of source code.

3.1 Formal and contextual understandings

This section elaborates our definition of understanding—the process of acquiring a working knowledge of an object². 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 complementary approaches centered around experience and situatedness

²Or, as Catherine Elgin puts it: "The cognitive competence involved in understanding is generally characterized as grasping. Propositional understanding involves grasping a fact; objectual understanding consists of grasping a range of phenomena. This seems right. But it is not clear what grasping is. I suggest that to grasp a proposition or an account is at least in part to know how to wield it to further ones epistemic ends" (Elgin, 2017)

outline an alternative tradition.

3.1.1 Between formal and informal

Understanding can be differentiated between the object of understanding and the means of understanding (Elgin, 2017). Here, we concern ourselves with the means of understanding, particularly as they are related to the development of computer science. As the science of information processing, the field is closely involved in the representation of knowledge, a representation that programmers then have to make their own.

Theoretical foundations of formal understanding

The theoretical roots of modern computation can be traced back to the early 20th century in Cambridge where both philosophers of logic and mathematicians, such as Bertrand Russell, Ludwig Wittgenstein, and Alan Turing, were working on the formalization of thinking. In their work, we will see that the formalization of knowledge operations are rooted in an operation representation of knowledge.

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 thoughts through clear, unambiguous language, a common conclusion could be reached without much effort³. The stakes presented are thus those of understanding what language really is, and how to use it effectively to, in turn, make oneself understood.

The demonstration that Wittgenstein undertakes is that language and logic are 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 pre-

³"Most questions and propositions of the philosophers result from the fact that we do not understand the logic of our language" (Wittgenstein, 2010).

cisions, going down levels of abstraction through increasing bulleted lists. Through the stylistic organization of his work, Wittgenstein hints at the possibility to consider language, itself pre-requisite for understanding, as a form of logic. This complements the older approach to consider logic as a form of language. In this sense, he stands in the lineage of Gottfried Leibniz's *Ars Combinatoria*, since Leibniz considers that one can formalize a certain language (not necessarily natural languages such as German or Latin), in order to design a perfectly explicit linguistic system. 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. His work also fits with that of Bertrand Russell and Alfred Whitehead who, in his *Principia Mathematica*, attempt to lay out a precise and convenient notation in order to express mathematical notations; similarly, Gottlieb Frege's work attempted to constitute a language in which all scientific statements could be evaluated, by paying particular attention to clarifying the semantic uncertainties between a specific sentence and how it refers to a concept (Korte, 2010).

Even though these approaches differ from, and sometimes argue with⁴, one another, we consider them to be part of a broad endeavour to find a

⁴See, ironically, Frege's critique of Russell and Whitehead's work, quoted in the Stanford Encyclopedia of Philosophy: "I do not understand the English language well enough to be able to say definitely that Russell's theory (*Principia Mathematica I*, 54ff) agrees with my theory of functions of the first, second, etc. levels. It does seem so. But I do not understand all of it. It

linguistic basis to express formal propositions through which one could establish truth-values.

Such works on formal languages as a means of knowledge processing 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 solving complex, abstract mathematical problems, manifested as a symbolic machine (Turing, 1936). Meaning formally expressed was to be mechanically processed.

The design of this symbol-processing machine, subsequently known as the Turing machine, is a further step in engaging with the question of knowledge processing in the mathematical 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), hitherto considered a basis for understanding, since solving a mathematical problem supposed, at the time, to be able to understand it.

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; or attention, comprehension, diction) (Turing, 2009). This conversational test, which has become a benchmark for machine intelligence, would naively imply the need for a machine to *understand* what is being said.

Throughout the article, Turing does not yet address the need for a purely formal approach of whether or not a problem can be translated into atomistic symbols, as we can imagine Leibniz would have had it which

is not quite clear to me what Russell intends with his designation $\phi!x\Box I$ never know for sure whether he is speaking of a sign or of its content." (Linsky & Irvine, 2022)

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 establish the truth of each of these sub-propositions, and then re-assembled to deduce the truth-value of the original proposition.

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 ways⁵.

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, discrete and *conceptually coherent* units (i.e. symbols). One can then decompose a thought in a list, and expect a counterpart to recompose this thought by perusing it. As a symbol system, lists establish clear-cut boundaries, are simple, abstract and discontinuous; incidentally, this makes it very suited to a discrete symbol-processing machine such as the computer (Depaz, 2023).

With these sophisticated syntactic systems developed a certain approach to cognition, as Turing clearly establishes a possibility for a digital

⁵Jack Goody develops the influence of notation on cognition: "[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)

computer to achieve the intellectual capacities of a human brain.

But as Turing focuses on the philosophical and moral arguments to the possibility for machines to think, he does address the issue of formalism in developing machine intelligence. Particularly, he acknowledges the need for intuition in, and self-development of, the machine in order to reach a level at which it can be said that the machine is intelligent. The question is then whether one is able to represent such concepts of intuition and development in formal systems. We now turn to the form of these systems, looking at how their form addresses the problem of clearly understanding and operating on mathematical and logical statements.

Being based on some singular, symbolical entity, the representation of logical calculus into lists and symbols, within a computing environment, becomes the next step in exploring these tools for thinking, in the form of programming languages. Considering understanding through a formal lens can then be confronted to the real world: when programmed using those formal languages, how much can a computer understand?

Practical attempts at implementing formal understanding

This putting into practice relies on a continued assumption of human cognition as an abstract, logical phenomenon. Practically, programming languages could logically express operations to be performed by the machine.

The first of these languages is IPL, the Information Processing Language, 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.

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*. As such, it proves to be 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. 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 (see 2.1.3).

This structure of Lisp 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 conceptually coherent parts (e.g. Phrase \rightarrow Noun-Phrase + Verb-Phrase \rightarrow 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 abstractly 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, *SHRDLU*'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, atomistic symbols.

A critique of formalism as the only way to model understanding was already developed in 1976 by Joseph Weizenbaum. Particularly, he argues that the machine cannot make a judgment, as judgments cannot be reduced to calculation (Weizenbaum, 1976). While the illusion of cognition might be easy to achieve, something he did in his development of early conversational agents, of which the most famous is *ELIZA*, the necessary inclusion of morals and emotion of the process of judging intrinsically limit what machines can do⁶. Formal representation might provide a certain appearance of understanding, but lacks its depth.

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 to abstract knowledge 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,

⁶Joseph Leighton considers judgment has a foundational aspect of understanding, which is the construction of operational knowledge: "knowledge begins in simple judgments, judgments of feeling or sentience, as yet devoid of explicit conceptual relations, but containing the germs of all higher order functions of thinking." (Leighton, 1907).

rather than self-contained propositions is, interestingly, rooted in his activity as a literary critic⁷.

Going further in the processes of the brain, McCullough indeed finds out, in another paper with Lettvin 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 and sensual 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 introductory 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)

So, while formal notation is able to enable digital computation, it proved to be limited when it came to accurately and expressively conveying meaning. This limitation, of being able to express formally what we

⁷Even 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)

understand intuitively (e.g. *what is a chair?*⁸) appeared as computers applications left the domain of logic and arithmetic, and were applied to more complex problem 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 and means of understanding, 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

With the publication of Wittgenstein's *Philosophical Investigations*, there was a radical posture change from one of the logicians whose work underpinned AI research. 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 of what language is. Rather than what knowledge is, he looks at how knowledge is acquired and used; while (formal) language 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 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 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, a branch of computation specifically focused on cognition, has shown some limits to this approach. Departing from formal languages, we now investigate how an embodied and situated agent can develop a certain sense of understanding.

⁸A question addressed by Joseph Kosuth in his conceptual artwork *One and Three Chairs*, 1965

Knowledge and situation

As hinted at by the studies of McCullough and Levitt, the process of understanding does not rely exclusively on abstract logical processes, but also on the processes involved in grasping a given object, such as, in their case, what is being seen. 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 universality 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 recurrently finds in the AI literature, such as Herbert Simon's *The Sciences of the Artificial* (Simon, 1996). In programming, to implement an algorithm means to manifest in concrete instructions, such that they are understood by the machine. The phrase thus refers to the gap existing between the statement of an idea, of an algorithm, and a procedure, and its concrete, effective and functional manifestation. This concept of implementation will show how context tends to complicate abstract understanding.

For instance, pseudo-code is a way to sketch out an algorithmic procedure, which might be considered agnostic when it comes to implementation details. At this point, the pseudo-code is halfway between a general idea and the specificity of the particular idiom in which it is inscribed. One can consider the pseudo-code in 38, 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

```

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

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), the programming language (e.g. JavaScript or Python), or any details about the human user drawing the circle.

Refuting the idea that pseudo-code, as abstracted representation, 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 full 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—programming languages only go so far in describing a problem domain without reducing such do-

main in a certain way.

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 particular, a symbol system does not need to be formal in order to act as a cognitive device. Logical notation exists along with music, painting, poetry and prose. 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), further explored in 4.1. What follows, argues Goodman, is that all these formal languages involve an act of *reference*. Through different means (exemplification, denotation, resemblance, representation), linguistic 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.

Despite the work of Shannon (Shannon, 2001) and its influence on the development of computer systems, communication, as the transfer of meaning from one individual to one or more other individuals, does not exclusively rely on the use of mathematical notation 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) (Marchand-Zañartu & Lauxerois, 2022). How specific domains, including visual arts and construction, engage in the relation between form and cognition is further addressed in chapter 4.

Going beyond formal understanding through logical notation, we have seen that there are other conceptions of knowledge which take into account the physical, social and linguistic context of the agent understand-

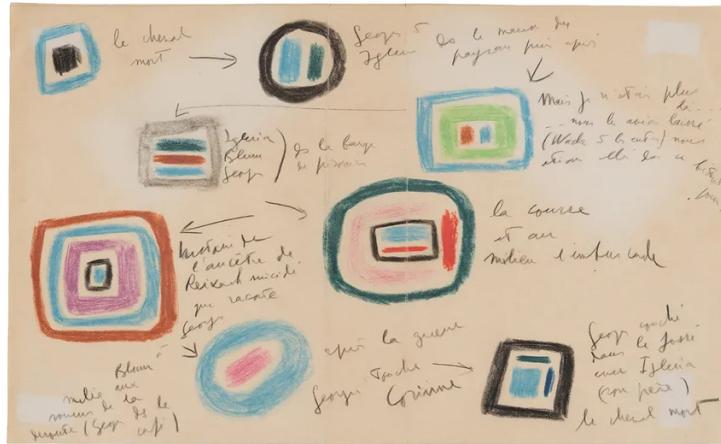


Figure 3.1: Tentative d'organisation visuelle pour le roman *La Route des Flandres*, années 1960 - Claude Simon, écrivain

ing, as well as of the object being understood. Keeping in mind the recurring concept of craft discussed in 2.3.3, complete this overview of understanding by paying attention to the role of practice.

Constructing knowledge

There are multiple ways to express an idea: one 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 explicated.

First of all, there is a separation between *knowing-how* and *knowing-that*; the latter, propositional knowledge, does not cover the former, practical knowledge (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 knowledge of something: to know *what* constructing a calculating machine entails and knowing *how* to construct such a machine. 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 un-specifiables, a knowing of more than you can tell. (Polanyi &

Grene, 1969)

Rooted in psychology, and therefore in the assumption of the embodiment 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 representations.

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 2.3.3, 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 here⁹. 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 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 artefact, which is accurate enough that it allows us to predict the behaviour of and to interact with such object.

Theories of how individuals acquire understanding (how they come to

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

know things, and develop operational and conceptual representations of things), have been approached from a formal perspective, and a contextual one. The rationalist, logical philosophical tradition from which computer science originally stems, starts from the assumption that meaning can be rendered unambiguous through the use of specific notation. Explicit understanding, as the theoretical lineage of computation, then became realized in concrete situations via programming languages.

However, the explicit specification of meaning fell short of handling everyday tasks which humans would consider to be menial. This has led us to consider a different approach to understanding, in which it is acquired through contextual and embodied means. Particularly, we have 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. Due to programing's *ad hoc* and bottom-up nature, attempts to formalize it have relied on the assumption that expert programmers have a certain kind of tacit knowledge (Soloway et al., 1982; Soloway & Ehrlich, 1984). The way in which this knowledge, which they are not able to verbalize, has been acquired and is being deployed, has long been an object of study in the field of software psychology.

Before our overview of what the psychology of programmers can contribute on the cognitive processes at play in understanding source code, we must first explicit in which ways software as a whole is a cognitively complex object.

3.2 Understanding computation

Software, computation and source code are all related components; respectively object, theory and medium. The ability to dematerialize software (from firmware, to packaged CDs, to cloud services) and the status of source code as intellectual property point to an ambiguous nature: it is both there and not there, idea and matter. This section makes explicit some of the affordances of software which make it a challenging object to grasp, in order to lay out what programmers are dealing with when they read and write source code.

In order to reconcile the different tensions highlighted in the various kinds of complexities that software exhibits, we first turn to an ontological stance. Particularly, we will develop on Norbay Irmak's proposal that software exists as an *abstract artifact*, simultaneously on the ideal, practical and physical planes, and see how Simondon's technical and aesthetic mode of existence can reconcile fragmented practice with unified totality.

We then shift to the practical specificities of software, particularly in terms of levels and types of complexity. This will highlight some of the properties that make it hard to understand, such as its relation to hardware, its relation to a specification, and its existence in time and space.

With this in mind, we will conclude this section by looking specifically at the source code representation of software, and at how programmers deploy strategies to understand it. Approaching it from a cognitive and psychological perspective, we will see how understanding software involves the construction of programming plans and mental models; the tools and helps used in order to construct them will be explicated in the next section.

3.2.1 Software ontology

Before we clarify what software complexity consists of, we first frame these difficulties in a philosophical context, more specifically the philosophy of

technology. We will investigate how these complexities can be seen as stemming from the nature of technology itself, and how this connects to an aesthetic stance. Before moving back to practical inquiries into how specific individuals engage with this nature, this section will help provide a theoretical background, framing technology as a relational practice, complementing other modes of making sense of and taking action on the world. This conceptual framework will start with an investigation into the denomination of software as an *abstract artifact*, followed by an analysis of technology as a specific mode of being, and concluding on how it is related to an aesthetic mode of being.

Software as abstract artifact

When he coins the phrase *abstract artifact*, Nurbay Irmak addresses software partly as an abstract object, similar in his sense to Platonic entities, and partly as a concrete object which holds spatio-temporal properties (Irmak, 2012). This is based on the fact that software requires an existence as a textual implementation, in the form of source code (Suber, 1988); it is composed of files, has a beginning (start) and an end (exit); but software also represents ideas of structure and procedure which go beyond these limitations of being written to a disk, having a compilation target or an execution time. Typically, the physical aspects of software (its manifestation as source code) can be changed without changing any of the ideas expressed by the software¹⁰.

Irmak complements Colburn's consideration of software as a *concrete abstraction*, an oxymoron which echoes the tensions denoted by the concept of the abstract artifact. He grounds these tensions in the distinction between a medium of execution (a—potentially virtual—machine) and a medium of description (source code). He considers that, while any high-

¹⁰In programming, this is called *refactoring*. This phenomenon can also be observed in natural languages, in which one can radically change a syntax without drastically changing the semantics of a sentence.

level programming language is already the result of layers of abstraction, such language gets reduced to the zeroes and ones input to the central processing unit (Colburn, 2000). Here, he sees the abstraction provided by languages ultimately bound to the concrete state of being of hardware and binary. And yet, if we follow along along his reasoning, these representations of voltage changes into zeroes and ones are themselves abstractions over yet another concrete, physical event. Concrete and abstract are recursively tangled properties of software.

Writing on computational artefacts, of which software is a subset, Raymond Turner formalizes this specificity of in a three-way relationship. Namely, abstract 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 but limited account of the place of source code within the definition of software: the Java implementation is itself a definition implemented in a specific bytecode, while arrays in Pascal are different abstractions than arrays implemented in C, etc. Nonetheless, source code is that which gives shape to the ideas immanent in software—through a process of concretization—and which hides away the details of the hardware—through abstraction. This metaphor of *abstract artifact* thus 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, like other artefacts, has a relation between its *functional* properties (i.e. purpose that are intended to be achieved through their use) and *structural* ones (both conceptual and physical configuration which are involved in the fulfilment of the functional purpose) (Turner, 2018). As such, it also belongs to the broader class of technology, and thus holds

some of the specificities of this lineage, into which we extend our inquiry.

Software as a relational object

The technological object underwent a first qualitative shift during the European Industrial Revolution, and a second one with the advent of computing technologies. The status of its exact nature is therefore a somewhat recent object of inquiry. Here, we will start from Gilbert Simondon's understanding of technology as a *mode*, in order to ultimately contrast it with the *aesthetic mode*.

According to Simondon, the technical object is a relation between multiple structures and the result of a complex operation of various knowledges (Simondon, 1958), some scientific, some practical, some social, some material. The technical object is indeed a scientific object, but also a social object and an artistic object at the same time. Differentiated in its various stages (object, individual, system), it is therefore considered as relational, insofar as its nature changes through its dependance, and its influence, on its environment.

Technology is a dynamic of organized, but inorganic matter (Stiegler, 1998). Following Latour, we also extend the conception of inorganized matter to include social influences, personal practices, and forms of tacit and explicit knowledges (Latour, 2007). 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(s) which shape and make use of it.

Technology is generally bound to practical matter, even though such matter could, under certain circumstances, take on a symbolic role of manifesting the abstract. This is the case of the compass, the printing press, or the clock. The clock, a technology which produces seconds, its action reached into another domain—that of mechanical operation on abstract ideas (Mumford, 1934). The domain of abstract ideas was hitherto reserved

to different modes than technology: that of religion and philosophy, and technology holds a particularly interesting relation with these two. According to Simondon, philosophy followed religion as a means of relating to, and making sense of, the abstract such the divine and the ethical. Tracing back the genesis of the technological object, he writes that the technical mode of existence is therefore just another mode through which the human can relate to the world, similar to the religious, the philosophical, and the aesthetic mode(Simondon, 1958).

Technical objects imply another mode of being, consequential to the recognition of the limitations of magic—humanity's primary mode of being. Technicity, according to Simondon, focuses on the particular, on the elements, *a contrario* to the religious mode of being, which finds more stability in a perspective of totality, rather than a focus on individuals¹¹.

This technical mode of existence, based on particulars, can nonetheless circle back to a certain totality through the means of induction; that is, deriving generals from the observed particulars. As such, technical thinking, as inverted religious thinking, stems from practice, but also provide a theory Technology, religion and philosophy are all, according to Simondon, combinations of a theory of knowledge and a theory of action, compensating for the loss of magic's totalizing virtues. While the religious, followed by the philosophical, approach from theory to deduce a practice, and thus lack grounding, technology reverses the process and induces theory from operations on individual elements.

Simondon complements the technical with the aesthetic mode, and as such counter-balances the apparent split between technics and religion by

¹¹"*La pensée technique a par nature la vocation de représenter le point de vue de l'élément ; elle adhère à la fonction élémentaire. La technicité, en s'introduisant dans un domaine, le fragmente et fait apparaître un enchaînement de médiations successives et élémentaires gouvernées par l'unité du domaine et subordonnées à elle. La pensée technique conçoit un fonctionnement d'ensemble comme un enchaînement de processus élémentaires, agissant point par point et étape par étape ; elle localise et multiplie les schèmes de médiation, restant toujours au-dessous de l'unité.*" (Simondon, 1958).

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 the aesthetic mode as a unifying network of relationships¹². He further argues that the aesthetic mode goes beyond taste and subjective preference, into a fundamental aspect of the way in which human beings relate to the world around them. An aesthetic object therefore acquires the property of being beautiful by virtue of its relationships, of its connections between the subject and the objective, between one's history and one's perceptions, and the various elements of the world, and the actions of the individual. Finally, the aesthetic thought when related to the technical 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. Ultimately, the aesthetic mode can therefore be seen as the revealing of a nexus of relationships found in its environment, highlighting the key-points of in the structure of the object¹³. How aesthetics enables a holistic thought through the use of sensual markers will be the subject of 4.

Computation, as a particular kind of computation, is thus both a theory and a practice, and can also be subject to an aesthetic impression. Particularly, one can think of computers as a form of technology through which *meaning is mechanically realized*¹⁴.

¹²"L'impression esthétique n'est pas relative à une œuvre artificielle ; elle signale, dans l'exercice d'un mode de pensée postérieur au dédoublement, une perfection de l'achèvement qui rend l'ensemble d'actes de pensée capable de dépasser les limites de son domaine pour évoquer l'achèvement de la pensée en d'autres domaines ; une œuvre technique assez parfaite pour équivale à un acte religieux, une œuvre religieuse assez parfaite pour avoir la force organisatrice et opérante d'une activité technique donnent le sentiment de la perfection." (Simondon, 1958)

¹³"Là apparaît l'impression esthétique, dans cet accord et ce dépassement de la technique qui devient à nouveau concrète, insérée, rattachée au monde par les points-clefs les plus remarquables" (Simondon, 1958).

¹⁴"Sans constraints of meaning or meaningfulness (i.e., some flavour of intentionality), computers would amount to nothing more than "machines"—or even, as I will ultimately argue, to "stuff": mere lumps of clay. Unless it recognizes meaningfulness as essential, even the most

Software is 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, manipulable notion of a state of a process, ever evolving around a fixed structure, and by changing the conceptual understanding of said world (Rapaport, 2005). Such examples of world related to the environment in which software exists, e.g. the social environment, or hardware environment, or the environment which has been recreated within software. David M. Berry investigates this encapsulation of world in his *Philosophy of Software*:

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-as-world is the material implementation of a proposed model, itself derived from a theory. It therefore primarily acts at the level of *episteme*, sometimes even limiting itself to it¹⁵. Paradoxically, it is only through

highly perfected theory of computation would devolve into neither more nor less than a generalized theory of the physical world. Sans some notion of efficacy of mechanism m, conversely, no limits could be either discerned or imposed on what could be computed, evacuating the notion of constraint, and hence of intellectual substance. Freed from all strictures of efficacy or mechanism from any requirement to sustain physical realizability, computation would become fantastic (or perhaps theistic): meaning spinning frictionlessly in the void." (Smith, 2016).

¹⁵Functional 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.

peripherals that software can act as a mechanical technology in the industrial sense of the word.

Along with software's material and theoretical natures (i.e. in contemporary digital computers, it consists of electrons, copper and silicium and of logical notations), another environment remains—that of the intent of the humans programming such software. Indeed, thinking through the function of computational artefacts, Turner states that it is *agency* which determines what the function is. He defines agency as the resolution of 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, to understand how it exists in multiple worlds, and how it represents the world, we need to give it meaning. To make sense of it, a certain amount of interpretation is required in relation to that of the computer's—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 human interpreters) (Rapaport, 2005). In his investigation into what software is, Suber corroborates:

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 have seen that while software can be given the particular status of an *abstract artifact*, these tensions are shared across technological objects, as they connect theory and practice. Technology, as a combination of a theory of knowledge and a theory of action, as an inter-

face to the world and a recreation of the world, is furthermore related to other modes of existence—and in particular the aesthetic mode. We have seen how Simondon suggests that the aesthetic mode has totalizing properties: through the sensual perception of perfected execution, it compensates technology's fragmented mode of existence.

What do these tensions and paradoxes look like in practice? In the next section, we examine more carefully the specific properties of software, and the complexities that this specific object entails. Specifically, we will see how software's various levels of existence, types of complexities, and kinds of actions and interpretations that it allows, all contribute to the cognitive hurdles encountered when attempting to understand software.

3.2.2 Software complexity

What is there to know about software? Looking at the skills that novel programmers have to develop as they learn their trade, one can include problem solving, domain modelling, knowledge representation, efficiency in problem solving, abstraction, modularity, novelty or creativity (Fuller et al., 2007). The variety of these skills and their connection to intellectual work—for instance, there is no requirement for manual dexterity or emotional intelligence—suggests that making and reading software is a complex endeavor.

Indeed, software exhibits several particularities, as it possesses several independent components which interact with each other in non-trivial, and non-obvious ways. In order to clarify those interactions, we start by looking at the different levels at which software exists, before turning to the different kinds of complexity which make software hard to grasp, concluding on its particular existence in time and space.

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)¹⁶. Specifically, there are three different types of software complexity that we look at: technical complexity, spatio-temporal complexity and modelling complexity.

Levels of software

Software covers a continuum from an idea to a bundled series of distinct binary marks. One of the essential steps in this continuum 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, instantiation, 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 reached¹⁷

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.

¹⁶As opposed to disorganized complexity, which are dealt with statistical tools.

¹⁷A 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.*". This quote is often mis-attributed to Richard P. Feynman or Albert Einstein, but has been traced to Benjamin Brewster, writing in the Yale Literary Magazine of 1882. (Investigator, 2018)

```

how to get the difference in character length between two words
store the first word in a variable
store the second word in a variable
store the difference between the number of characters in the first
→ word
and the number of characters in the second word
print the difference to the console

```

Listing 39: Example of a program text represented in pseudo code. See 40, 41 and 42 for lower level representations.

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 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 general 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. To illustrate the chain of material levels at which software exist, the series of listings in 39, 40, 41 and 42 perform the exact same function of implementing a FILL ME algorithm, respectively in pseudo code, in C, in Assembly and in bytecode.

The gradient across 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

```

#include <string.h>
#include <stdio.h>

int main(){
    char* a_word = "Gerechtigkeit";
    char* an_unword = "Menschenmaterial";

    int difference = strlen(a_word) - strlen(an_unword);

    printf("%d", difference);

    return 0;
}

```

Listing 40: Example of a program text represented in a high level language. See 39 for a higher level representation and 41 and 42 for lower level representations.

```

push    %rbp
mov     %rsp,%rbp
movl   $0xa,-0xc(%rbp)
movl   $0x2,-0x8(%rbp)
mov    -0xc(%rbp),%eax
sub    -0x8(%rbp),%eax
mov    %eax,-0x4(%rbp)
mov    $0x0,%eax
pop    %rbp
ret

```

Listing 41: Example of a program text represented in an Assembly language. See 39 and 40 for a higher level representation and 42 for a lower level representation.

1119:	55
111a:	48 89 e5
111d:	c7 45 f4 0a 00 00 00
1124:	c7 45 f8 02 00 00 00
112b:	8b 45 f4
112e:	2b 45 f8
1131:	89 45 fc
1134:	b8 00 00 00 00
1139:	5d
113a:	c3

Listing 42: Example of a program text represented in bytecode. See 39, 40 and 41 for higher level representations.

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. For instance, the machine on which a given program text is executed can be a *virtual machine* or, conversely, a real machine managing virtual memory.

In any case, these are only the technical components underpinning software, its specifications and formalizations. Another dimension of complexity is introduced by the fact that software is supposed to interact

with entities that are not already formalized nor quantized, such as physical reality and its actors.

Spatio-temporal complexity

A rough way of describing computers is that they are extremely stupid, but extremely fast (Muon Ray, 1985). The use of programming language is therefore a semantic translation device between a natural problem, the formalization of the problem in such a language, and the binary expression of the program which can be executed by the CPU at very high speeds.

This very high speed of linear execution involves another dimension to be taken into account by programmers. 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 a function call) (Lando et al., 2007). Whether something changes over time, and when such a thing changes becomes an additional cognitive load for the programmer reading and writing source code, a load which can be alleviated by data types (such as the `const` keyword, marking a variable as unchangeable), or by aesthetic marks (such as declaring a variable in all capital letters to indicate that it *should* not change).

Temporal complexity relates to the discrepancy between the way the computer was first thought of —i.e. as a Turing machine which operates linearly, on a one-dimensional tape—and further technological developments. The hardware architecture of a computer, and its specification as a Turing machine involve the ability for the head of the machine to jump at different locations. This means 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 particularly obvious in Ben Fry's *Distellamap* series of visualizations of source code (3.2 represents the execu-

tion of the source code for the arcade game Pac-Man)¹⁸.

Furthermore, the machine concept of time is different from the human concept, and different machines implement different concepts. For instance, operations can be synchronous or asynchronous, thus positing opposite frames of reference, since the only temporal reference is the machine itself¹⁹. While humans have somewhat intuitive conceptions of time as a linearly increasing dimension, computer hardware actually includes multiple clocks, used for various track-keeping purposes and structuring various degrees of temporality (Mélès, 2017).

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 transformed 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. 1/18000000th of a second on a 3.0 Ghz CPU machine) becomes crucial to ensuring the correct behaviour of the software.

Conversely, the locii of the execution of software creates contributes to those issues. Even at its simplest, a program text does not necessarily exist as a single file, and is never read linearly. Different parts can be re-edited and re-arranged to facilitate the understanding of readers²⁰. Mod-

¹⁸This is the kind of convoluted trace of execution which led to Edsger W. Dijkstra's statement on the harmfulness of such jumps on the cognitive abilities of programmers, especially the GOTO statement Dijkstra (1968)

¹⁹Baptiste Mélès analyzes this temporal ontology of the computer: "The clock's name is deceptive: even if, viewed from the outside, its operation is based on the regularity of a physical phenomenon—typically the oscillation of a quartz crystal when an electric current passes through it—it does not tell the time, as though its job were simply to measure it. Rather, it tells the machine what time it is. From the machine's point of view, this is not a component that reads the time, but writes it." (Mélès, 2017).

²⁰For instance, John Lions's *Commentary On UNIX version 6* includes extensive editorial

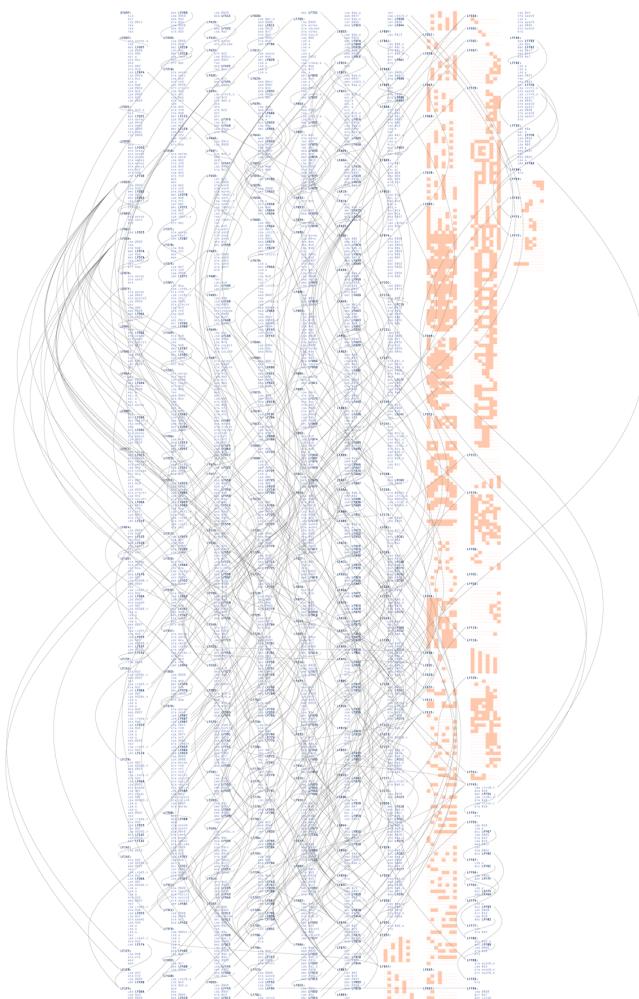


Figure 3.2: Visualization of the execution of Pac-Man's source code

ern programming languages also have the feature of including other files, not directly visible to the user. The existence of those files have a textual manifestation, such as the `#include` line in C or `import` in Python, but the contents of the file can remain elusive.

Where exactly these files exist is not always immediately clear, as their reference by name or by Uniform Resource Locator (URL) can obfuscate whether or not a file exists on the current machine. As such, 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). Facilitating navigation between files through the references that files hold to one another is one way that the tools of the programmers alleviate cognitive burden, as we will see in 3.3.2.

Additionally, time and space in computation can interact in unexpected ways, and 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). When and where a certain action takes place becomes particularly hard to follow.

Modelling complexity

Modeling complexity addresses the hurdles in translating a non-discrete, non-logical object, event, or action, into a discrete, 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 somewhat

work to make sense of the textual matter written by Ken Thompson and Dennis Ritchie (Lions, 1996)

straightforward from a human perspective become more intricate once the time for implementation has come.

This translation process involves the development of *models*; these are abstract descriptions of the particular entities which are considered to be meaningful in the problem domain. The process of abstracting elements of the problem domain into usable computational entities is an essential aspect of software development, as it composes the building blocks of software architectures (see 2.1.1 for discussion of software architects). Abstraction encompasses different levels, at each of which some aspect of the problem domain is either hidden or revealed, and finding the right balance of such showing or hiding in those models does not rely on explicit and well-known rules, but rather on cognitive principles. Starting from the observation that there no generalizable rules for modelling classes in computer science, Parsons and Wand suggest that cognitive principles can be a productive way forward²¹. They base their proposal on the theories of Lakoff and Johnson, insofar as metaphors operate cognitively by mapping two entities abstracting at the same level; such a tool for understanding is further explored in 3.3.1.

For a banking system, this might involve a `Client` model, an `Account` model, a `Transfer` model and a `Report` model, among others. The ability to represent a `Client` model at a productive abstraction level is then further complicated by the conceptual relations that the model will hold with other models. Some of these relations can be made explicit or implicit, and interact in unexpected ways, since they differ from what our personal conception of what a `Client` is and of what it can do²².

²¹"The classes we form reflect our experience with things. That is, we form our concepts by abstracting our knowledge about instances. Furthermore, the concepts we use are not chosen arbitrarily. Concept theory proposes that classification is governed by the two primary functions of concept formation in human survival and adaptation: cognitive economy and inference." (Parsons & Wand, 1997).

²²"In the process of modelling some part of the world in an object-oriented fashion the focus is on identifying concepts and their mutual relations and then describing these by means

Working at the "right" layer of abstraction then becomes a contextual choice of reflecting the problem accurately, taking into account particular technical constraints, or the social environment in which the code will circulate. For instance, choosing to represent a color value as a three-dimensional vector might be efficient and elegant for an experienced programmer, but might prove confusing to beginner programmers. The key aspect of being a triplet might be lost to someone who focuses on the suggested parallels between points in space and a shade of red.

Let us consider a simple abstraction, such as having written publications, composed of three components: the name of an author, the date of publication, and the content of the publication. This apparently useful and practical abstraction becomes non-straightforward once the system that uses it changes in scale. With a hundred publications, it is easy to reason about them. With a million publications, the problems themselves start to change, and additional properties such as tags, indexes or pages should be considered in modelling the publication for the computer (Cities, 2022).

The aphorism "*All models are wrong, but some are useful*" (Box, 1976) captures the ambiguity of abstraction of a model from real-world phenomena. The aim of a model is to reduce the complexity of reality into a workable, functional entity that both the computer and the programmer can understand. This process of abstraction is the result of judging which parts of a model are essential, and which are not and, as we have seen in 3.1.1, judgments involve a certain amount of subjectivity (Weizenbaum, 1976).

Ultimately, the concrete representation of a model involves concrete syntax through the choice of data types, the design of member functions and the decision to hide or reveal information to other models. Which individual tokens and which combination of tokens are used in the repre-

of classes and associations between them. Using existing methods and notation we usually describe all the classes and the corresponding associations between them in one, flat model, despite the fact that these are typically at different levels of detail. Consequently the description often appears confusing and disorganized" (Kristensen, 1994)

sentation process then contribute to communicate the judgment that was made in the abstraction process.

Software involves, through programming languages, the expression of human-abstracted models for machine interpretation, which in turn is executed at a scale of time and space that are difficult to grasp for individuals. These properties make it difficult to understand, from conception to application: software in the real-world go through a process of implementation of concepts that lose in translation, interfacing the world through discrete representations, and following the execution of these representations through space and time. Still, source code is the material representation of all of these dynamics and the only point of contact between the programmer's agency and the machine execution and, as such, remains the locus of understanding. Programmers have been understanding software as long as they have been writing and reading it. We now turn to the attempts at studying the concrete cognitive processes deployed by source code readers and writers as they engage meaningfully with program texts.

3.2.3 The psychology of programming

In practice, programmers manage to write, read and understand source code as a pre-requisite of producing reliable source code. Being able to write a program has for effective pre-requisite a thorough understanding of the problem, intent and platform, making the programming activity a form of applied understanding²³.

How programmers deal with such a complex object as software has been a research topic which appeared much later than software itself. The field of software psychology aims at understanding how programmers process code, and with which level of success, and under which conditions. How do they build up their understanding(s), in order to afford appropri-

²³"We understand what we are able to program." (Ershov, 1972)

ate 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 answering these questions, we will see how the process of understanding a program text is akin to constructed a series of mental models, populating a cognitive map.

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; Weinberg, 1998). 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, and it is hard to clearly establish the impact of each of these. Second, program comprehension is strongly influenced by practice—the skill level of the programmer therefore also influences experimental conditions²⁴. Third, these early studies have found that programmers have organized knowledge bases, if informal and immaterial. 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 mental models.

Marian Petre and Alan Blackwell attempted in their 1992 study 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. 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

²⁴Weinberg establishes a connection between value and the appropriate level of skill application: "*The moral of this tale—and a hundred others like it—is that each program has an appropriate level of care and sophistication dependent on the uses to which it will be put. Working above that level is, a way, even less professional than working below it. If we are to know whether an individual programmer is doing a good job, we shall have to know whether or not he is working on the proper level for his problem.*" (Weinberg, 1998)

oneself, or someone else), and the code now needs to be read.

The main conclusion of their study is that, beyond the fact that each programmer had slightly different descriptions of their mental process, there are 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.

Francoise Détienne, in her study of how computer programmers design and understand programs (Detienne, 2001), 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 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. To do so, they seem to resort to spatial activities, such as *chunking* and *tracking* (Cant et al., 1995), thus hinting at a need to delimitate some cognitive objects with a material metaphor, and connecting those concepts with a spatial metaphor.

In programming, within a given context—which includes 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), Détienne nonetheless highlights some differences with natural language understanding. In program texts, she finds, 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 (Wiedenbeck, 1991; Koenemann & Robertson, 1991). 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 specifically is how the specific surface structure in programming results in the understanding of the deep structure—in other terms, what is the connection be-

tween source code syntax, programmer semantics and program behavior.

Détienne's work ushers in the concept of a mental model as means of understanding in programmers, which proved to be a fruitful, if not settled field of research. Mental models are a dynamic representation formed in working memory as a result of using knowledge from long term memory and the environment (Cañas & Antolí, 1998). As such, they are a kind of internal symbolic representation of an external reality, are a rigorous, personal and conceptual structure. They are related to knowledge, since the construction of accurate and useful mental models through the process of understanding is shaped by, and also 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 than a metaphor.

Further research on mental model acquisition have established a few parameters which influence the process. First, programmers have a background knowledge that they activate through the identification of specific recurring patterns in the source code, confirming Détienne characterization of the roles of beacons. Second, mental models seem to be organized either as a layered set of abstractions, providing alternative views of the system as needed, or as a groups or sets of heuristics. Finally, programmers use both top-down processes of recognizing familiar patterns, they also make use of bottom-up techniques to infer knowledge from which they can then construct or refine a mental model (Heinonen et al., 2023).

Epistemic actions, the kinds of actions which change one's knowledge of the object on which the actions are taken, contribute to reducing the kinds of complexities involved with software. Concretely, this involves refining the idea that one has of the software system at hand, by comparing

the result of the actions taken with the current state of the idea(s) held. 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)

Since epistemic actions rely on engaging with a text, at the syntax and semantics level, it has often been assumed by programmers and researchers that reading and writing code is akin to reading and writing natural language. 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 text* 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, through a particular syntactic configuration. 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 ²⁵, 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, mathematic 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 & 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 interpreted is the intent of the author(s). Such a tension between the computer's position as an extremely fast executer and the programmer's position as a cognitive agent is summer up by Niklaus Wirth in *Beauty Is Our Business*, Dijkstra's *festschrift*: "*What the computer interprets, I wanted to understand.*" (Wirth, 1990).

One key aspect of the acquisition process seems to be mapping or linking features of the actual target system to its mental representation. The result of have been referred to as cognitive maps or knowledge maps. Here

The complexities of software are echoed in how programmers evoke their experience of either designing or, comprehending code. They have shown to use multiple cognitive abilities, without being strictly limited to narrative, or mathematic frames of understanding, and making use of no-

²⁵More often than not, a pestering bug

tions of scale and focus to disentangle complexity. For the remaining section of this chapter, we will focus on two specific means that contribute to this process of building a mental model of software-as-source code. Based on the reports that programmers use mental images and play with dynamic mental structures to comprehend the functional and structural properties of software, we can now say that understanding of a program text involves the construction of mental models. This happens through a process of mapping textual cues with background knowledge at various layers of abstraction, resulting in a cognitive cartography allowing for an program text to be made intelligible, and thus functional, to the programmer.

We conclude this chapter with a look at two practical ways in which sense is made from computational systems. From a linguistic perspective, we look at the role that metaphors play in translating computational concepts into ones which can be grasped by an individual. From a technical perspective, we start from the role of layout (indentation, typography) to develop on the concept of extended cognition to see how understanding is also located in a programmers' tools.

3.3 Means of understanding

Drawing on the ambivalence of software's existence—both concrete and abstract—as well as on the various way that software is a complex cognitive object to grasp, we now investigate the means deployed to render it meaningful to an individual. As we have seen in empirical studies, programmers resort to textual perusing in order to build up mental models.

In this section, we look at the particular syntactic tokens that are used to metaphorically convey the meaning of a computational element, as well

as the medium through which the medium is perused—via integrated development environments. This will conclude our inquiry into software's complexities and into how metaphors and textual manipulation facilitate the construction of mental models, before we inquire specifically about the ways in which aesthetics play a role in this process.

3.3.1 Metaphors in computation

Our understanding of metaphors relies on the work of George Lakoff and Mark Johnson²⁶ due to their requalification of the nature and role of metaphor beyond an exclusively literary role. While Lakoff and Johnson's approach to the conceptual metaphor will serve a basis to explore these linguistic devices as a cognitive means across software and narrative, we also argue that Ricoeur's focus on the tension of the *statement* rather than primarily on the *word* will help us better understand some of the aesthetic manifestations and workings of software metaphors. Following a brief overview of their contributions, we then examine the various uses of metaphor in software, from end-users to programmers.

Theoretical background

We start from the most commonly used definition of metaphor: that of labeling one thing in terms of another, thereby granting additional meaning to the subject at hand. Our approach here will also bypass some of the more minute distinctions of literary devices made between metonymy (in which the two things mentioned are already conceptually closely related), comparison (explicitly assessing differences and similarities between two things, often from a value-based perspective) and synecdoche (representing a whole by a subset), as we consider these all subsets of the class of metaphors.

²⁶We also develop from Ricoeur's conception of metaphors in 4.2.1.

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, but which can nonetheless suggest the property of dynamic evolution. These source cognitive structures possess *schemas*, which are defined enough to not be mistaken for something else, but broad enough to allow for multiple variants of itself to be applied to various targets, providing both reliability and diversity (Lakoff & Johnson, 1980). As we will see below, their approach allows us to focus not just on textual objects, but on the vast range of metaphors used also in computing-related environments. Given that the source of the metaphor should be well-grounded, with as little invariability as possible, in order to qualify a potentially ill-defined target domain, we see how this is a useful mechanism to provide an entrypoint to end users and novice programmers to grasp new or foreign concepts.

Starting with the role of metaphors manifested in expressions such as *the desktop*, *the mouse*, or *the cloud* for end-users, we will then turn to the programmers relationships to their environment as understood metaphorically. The relationship between poetic metaphor and source code will be developed in 5.2.3; with the topic of syntax and semantics in programming languages in 5.1.1, we will see that metaphor-induced tensions can be a fertile ground for poetic creation through aesthetic manifestations.

Metaphors for end-users

It is interesting to consider that the first metaphor in computing might be concomitant with the first instance of modern computing—the Turing *machine*. While Turing machines are widely understood as being manifested into what we call digital computers (laptops, tablets, smartphones, etc.), and thus definitely within the realm of mechanical devices, the Turing machine is not strictly a machine *per se*. Rather, it is more accurately defined as a mathematical model which defines an abstract machine. Indeed, as we saw in 3.2.1, computers cannot be proven or assumed to be machines, because their terminology comes from logic, textual, or discursive traditions (e.g. reference, statement, names, recursion, etc.) and yet they are still *built* (Smith, 1998). Humans can be considered Turing machines (and, in fact, one of the implicit requirements of the Turing machine is that, given enough time and resources, a human should be able to compute anything that the Turing machine can compute), and non-humans can also be considered Turing machines²⁷. Debates in computer science related to the nature of computing (Rapaport, 2005) have shown that computation is far from being easily reduced to a simple mechanical concern, and the complexity of the concept is perhaps why we ultimately revert to metaphors in order to better grasp them.

As non-technical audiences came into contact with computation through the advent of the personal computer, these uses of metaphors became more widespread and entered public discourse once personal computing became available to ever larger audiences. With the release of the XEROX Star, features of the computer which were until then described as data processing were given a new life in entering the public discourse. The Star was seminal since it introduced technological innovations such as a bitmapped display, a two-button mouse, a window-based display includ-

²⁷See research in biological computing, using DNA and protein to perform computational tasks (Garfinkel, 2000)

ing icons and folders, called a desktop. In this case, the desktop metaphor relies on previous understanding of what a desktop is, and what it is used for in the context of physical office-work; since early personal computers were marketed for business applications, these metaphors built on the broad cognitive structures of the user-base in order to help them make sense of this new tool.

Paul DuGay, in his cultural study of the Walkman, makes a similar statement when he describes Sony's invention, a never-before-seen compound of technological innovations, in terms of pre-existing, and well-established technologies (du Gay et al., 2013). The icon of a floppy disk for writing data to disk, the sound of wrinkled paper for removing data from disk, the designation of a broad network of satellite, underground and undersea communications as a cloud, these are all metaphors which help us make a certain sense of the broad possibilities brought forth by the computing revolution (Wyatt, 2004). Even the *clipboard*, presented to the user to copy content across applications, does not believe at all like a real clipboard (Barrera, 2022).

The work of metaphors takes on an additional dimension when we introduce the concept of interfaces. As permeable membranes which enable (inter)actions between the human and the machine, they are essential insofar as they render visible, and allow for, various kinds of agency, based on different degrees of understanding. Departing from the physically passive posture of the reader towards an active engagement with a dynamic system, interfaces highlight even further the cognitive and (inter)active role of the metaphor.

These depictions of things-as-other-things influence the mental model which we build of the computer system we interact with. For instance, the prevalent windows metaphor of our contemporary desktop and laptop environments obfuscates the very concrete action of the CPU (or CPUs, in the case of multi-core architecture) of executing one thing at a time, except at speeds which cannot be intuitively grasped by human perception.

Alexander Galloway's work on interfaces as metaphorical representations suggests a similar concern of obfuscation, as he recall Jameson's theory of cognitive mapping. Jameson uses it in a political and historical context, defining that a cognitive mapping is a "*a situational representation on the part of the individual subject to that vaster and properly unrepresentable totality which is the ensemble of society's structures as a whole*" (Jameson, 1991). To do so, Jameson starts from Lynch's inquiry into the psychic relation to the built environment (which we will return to in 4.3), insofar as a cognitive map is necessary to deploy agency in a foreign spatial environment, an environment which Jameson associates with late capitalism.

Galloway productively deploys this heuristic in the context of interfaced computer work: cognitive mapping is the process by which the individual subject situates himself within a vaster, unrepresentable totality, a process that corresponds to the workings of ideology²⁸. Here, we can see how metaphors can act as both cognitive tools to make sense of objects, but also as obfuscating devices to cloak the reality of the environment²⁹. The cognitive processes enable by metaphors help provide a certain sense of the unthinkable, of that which is too complex to grasp and therefore must be put into symbols (words, icons, sounds, etc.).

Nielsen and Gentner develop on some challenges that arise when one uses metaphors not just for conceptual understanding, but for further conceptual manipulation. In *The Anti-Mac Interface*, they point out that differences in features between target domain and source domain are inevitable. For instance, a physical pen would be able to mark up any part of a physical form, whereas a tool symbolized by a pen icon on a document editing software might restrict an average user to specific fields on the form. Their study leads to assess alternatives to one kind of interface³⁰, in or-

²⁸The relation between which has been explored by Galloway, Chun, Holmes and others, and is particularly apparent in how an operating system is designated in French: *système d'exploitation*, an exploitation system (Galloway, 2006; Chun, 2005).

²⁹Indeed, data centers are closer to mines than to clouds.

³⁰In their study, they refer to the one designed by Apple for the Macintosh in the 1990s.

der to highlight how a computer system with similar capabilities (both being Turing-complete machines), could differ in (a) the assumptions made about the intent of the user, (b) the assumptions made about the expertise level of the user and (c) the means presented to the user in order to have them fulfill their intent (Gentner & Nielsen, 1996).

Moving away from *userland*, in which most of these metaphors exist, we now turn to examine the kinds of metaphors that are used by programmers and computer scientists themselves. Since the sensual reality of the computer is that it is a high-frequency vibration of electricity, one of the first steps taken to productively engage with computers is to abstract it away. The word *computer* itself can be considered as an abstraction: originally used to designate the women manually inputting the algorithms in room-scale mainframes, the distinction between the machine and its operator was considered to be unnecessary. The relation between metaphor and abstraction is a complex one, but we can say that metaphorical thought requires abstraction, and that the process of abstraction ultimately implies designating one thing by the name of another (a woman by a machine's, or a machine by a woman's), being able to use it interchangeably, and therefore lowering the cognitive friction inherent to the process of specification, freeing up mental resources to focus on the problem at hand (Chun, 2005).

Metaphors are implicitly known not to be true in their most literal sense. Max Black in *Models and Metaphors* argues that metaphors are too loose to be useful in analytic philosophy but, like models they help make concepts graspable and render operation to the computer conceivable, independently of the accuracy of the metaphor to depict the reality of the target domain.

Abstraction, metaphors and symbolic representations are therefore used tools when it comes to understanding some of the structures and objects which constitute computing and software, in terms of trying to represent to ourselves what it is that a computer can and effectively does, and in terms of explaining to the computer what it is we're trying to operate on

(from an integer, to a non-ASCII word, to a renewable phone subscription or to human language).

When they concern the work of programmers, these tools deployed during the representational process differ from conventional or poetic metaphors insofar as they imply some sort of productive engagement and therefore empirically verifiable or falsifiable. These models are means through which we aim at constructing the conceptual structures on which metaphors also operate, and explicit them in formal symbol systems, such as programming languages.

Programmer-facing metaphors

Programmers, like users, also rely heavily on metaphors to project meaning onto the entities that they manipulate. Fundamentally, the work of these metaphors are not different from the ones that operate in the public discourse, or at the graphical interface level; nonetheless, they show how they permeate computer work in general, and source code in particular.

Perhaps one of the first metaphors a programmer encounters when learning about the discipline is the one stating that a function is like a kitchen recipe: you specify a series of instructions which, given some input ingredients (arguments), result in an output result (return value). However, the recipe metaphor does not allow for an intuitive grasping of *overloading*, the process through which a function can be called the same way but do things with different inputs. Similarly, the use of the term *server* is conventionally associated and represented as a machine sending back data when asked for it, when really it is nothing but an executed script or process running on said machine.

Another instance of symbolic use relying on metaphorical interpretation can be found in the word *stream*. Originally designating a flow of water within its bed, it has been gradually accepted as designating a continuous flow of contingent binary signs. *Memory*, in turn, stands for record,

and is stripped down of its essentially partial, subjective and fantasized aspects usually highlighted in literary works (perhaps *volatile memory* gets closer to that point). Finally, *objects*, which came to prominence with the rise of object-oriented programming, have only little to do with the physical properties of objects, with no affordance for being traded, for acting as social symbols, for gaining intrinsic value, but rather the word is used as such for highlighting its boundedness, states and actions, and ability to be manipulated without interfering with other objects.

Most of these designations, stating a thing in terms of another aren't metaphors in the full-blown, poetic sense, but they do, agains, hint at the need to represent complex concepts into humanly-graspable terms, what Paul Fishwick calls *text-based aesthetics* (Fishwick, 2006). The need for these is only semantic insofar as it allows for an intended interaction with the computer to be carried out successfully—e.g. one has an intuitive understanding that interrupting a stream is an action which might result in incompleteness of the whole. This process of linguistic abstraction doesn't actually require clear definitions for the concepts involved. For instance, example of the terminology in modern so-called cloud computing uses a variety of terms stacked up to each other in what might seem to have no clear *denotative* meaning (e.g. Google Cloud Platform offers *Virtual machine compute instances*), but nonetheless have a clear *operative* meaning (e.g. the thing on which my code runs). This further qualifies the complexity of the sense-making process in dealing with computers: we don't actually need to truly understand what is precisely meant by a particular word, as long as we use it in a way which results in the expected outcome. That being said, there is a certain correlation between skills and metaphors: the more skilled a programmer is, the less they resort to metaphors and they more they consider things "as they are" (McKeithen et al., 1981).

This need to re-present the specificities of the machines has also been one of the essential drives in the development of programming languages. Since we cannot easily and intuitively deal with binary notation to rep-

resent complex concepts, programming helps us deal with this hurdle by presenting things in terms of other things. Most fundamentally, programming languages represent binary signs in terms of English language (e.g. from binary to Assembly, see 3.2.2). This is, again, by no means a metaphorical process, but rather an encoding process, in which tokens are being separated and parsed into specific values, which are then processed by the CPU as binary signs.

Still, this abstraction layer offered by programming languages allowed us to focus on *what* we want to do, rather than on *how* to do it. The metaphorical aspect comes in when the issue of interpretation arises, as the possibility to deal with more complex concepts required us to grasp them in a non-rigorous way, one which would have a one-to-one mapping between concepts. Allen Newell and Herbert A. Simon, in their 1975 Turing Award lecture, offer a good example of symbolic manipulation relates inherently to understanding and interpretation:

*In none of [Turing and Church's] systems is there, on the surface,
a concept of the symbol as something that designates.*

The complement to what he calls the work of Turing and Church as automatic formal symbol manipulation is to be completed by this process of *interpretation*, which they define simply as the ability of a system to designate an expression and to execute it. We encounter here one of the essential qualities of programming languages: the ambivalence of the term *interpretation*. A machine interpretation is clearly different from a human interpretation: in fact, most people understand binary as the system comprised of two numbers, 0 and 1, when really it is interpreted by the computer as a system of two distinct signs (red and blue, Alex and Max, hot and cold, etc.). To assist in the process of human interpretation, metaphors have played a part in helping programmers construct useful mental representations related to computing. Keywords such as `loop`, `wildcard`, `catch`, or `fork` are all metaphorical denominations for computing processes.

These metaphors can go both ways: helping humans understand computing concepts, and to a certain extent, helping computers understand human concepts. This reverse process, using metaphors to represent concepts to the computer, something we touched upon in 3.2.2, brings forth issues of conceptual representation through formal symbolic means. The work of early artificial intelligence researchers consisted not just in making machines perform intelligent tasks, but also implies that intelligence itself should be clearly and unambiguously represented. The work of Terry Winograd, for instance, was concerned with language processing—that is, interpretation and generation. Through his inquiry, he touches on the different ways to represent the concept of language in machine-operational terms, and highlights two possible representations which would allow a computer to interact meaningfully with language (Winograd, 1982). He considers a *procedural* representation of language, one which is based on algorithms and rules to follow in order to generate an accurate linguistic model, and a *declarative* representation of language, which relies on data structures which are then populated in order to create valid sentences. At the beginning of his exposé, he introduces the historically successive metaphors which we have used to build an accurate mental representation of language (language as law, language as biology, language as chemistry, language as mathematics). As such, we also try to present language in other terms than itself in order to make it actionable within a computing environment, in a mutually informing movement.

Metaphors are used as cognitive tools in order to facilitate the construction of mental models of software systems. The implication of spatial and visual components in mental models already highlighted by Lakoff and Johnson, and pointed out through the psychology experiments on programmers allow us to turn to metaphors as an architecture of thought (Forsythe, 1986). Metaphors operate cognitively, Lakoff and Johnson argue, because of the embodiment which underpins every individual's perception. Therefore, such a use of metaphors points to the spatial nature

of the target domain, something already suggested by the concept of mapping in 3.2.3. Complementing the semantic structure of metaphor, we now turn to another conception of space in program texts: the syntactic structure of source code, upon which another kind of tools can operate.

3.3.2 Tools as a cognitive extension

Metaphors make use of their semantic properties in order to allow users to build an effective mental model of what the system is or does; as the result, they allow programmers to build up hypotheses and take epistemic actions to see whether their mental model behaves as expected. Some of the keywords of programming languages are thus metaphorical. However, one can also make use of the syntactical properties of source code in order to facilitate understanding differently. We see here how these tools take part in a process of extended cognition.

We have seen in 3.3.1 how interfaces decide on the way the abstract entities are represented, delimited and accessed. They can nonetheless also go beyond representation in order to alleviate cognitive load through technical affordances, by providing as direct access as possible to the underlying abstract entities represented in source code's structure.

Looking at it from the end-user's perspective, there is software which focuses on knowledge acquisition through direct manipulation. For instance, Ken Perlin's *Chalktalk* focuses on freehand input creation and programmatic input modification in order to explore properties and relations of mathematical objects (e.g. geometrical shapes, vectors, matrices) (Perlin, 2022), while Brett Victor's *Tangled* focuses in a very sparse textual representation of a dynamic numerical model. The epistemic actions taken within this system thus consists in manipulating the numbers presented in the text result in the modification of the text based on these numbers (Victor, 2011b,a).

For programmers, the kind of dedicated tool used to deal with source

code is called *Integrated Development Environment* (IDE). With a specific set of features developing over time, and catered to the needs and practices of programmers, IDEs cover multiple features to support software writing, reading, versioning and executing—operations which go beyond the simple reading of text (Kline & Seffah, 2005).

One of the first interfaces for writing computer code included the text editor called *EMACS* (an acronym for *Editor MACroS*), with a first version released in 1976. Containing tens of thousands of commands to be input by the programmer at the surface-level in order to affect the deeper level of the computing system, *EMACS* allows for remote access of files, modeful and non-linear editing, as well as buffer-based manipulation *Vim* (Greenberg, 1996). This kind of text editor acts as an interfacing system which allows for the almost real-time manipulation of digitized textual objects.

While software such as *EMACS* and *Vim* are mostly focused on productivity of generic text-editing, other environments such as *Turbo Pascal* or *Maestro I* focused specifically on software development tasks in a particular programming language in software such as the Apple WorkShop (1985) (West, 1987), or the Squeak system for the Smalltalk programming language (Ingalls et al., 1997). These tools take into account the particular attributes of software to integrate the tasks of development (such as linking, compiling, debugging, block editing and refactoring) into one software, allowing the programmer to switch seamlessly from one task to another, or allowing a task to run in parallel to another task (e.g. indexing and editing). Kline and Seffah state the goals of such IDEs: "*Such environments should (1) reduce the cognitive load on the developer; (2) free the developer to concentrate on the creative aspects of the process; (3) reduce any administrative load associated with applying a programming method manually; and (4) make the development process more systematic.*" (Kline & Seffah, 2005).

One of the ways that IDEs started to achieve these goals was by developing more elaborated user-interfaces, involving more traditional concepts of aesthetics (such as shape, color, balance, distance, symmetry). At the

surface level, concerned only with the source code's representation, and not with its manipulation. Indeed, since the advent of these IDEs, studies have demonstrated the impact that such formal arrangement has on program comprehension(Oman & Cook, 1990b; Oliveira et al., 2022). Spacing, alignment, syntax highlighting and casing are all parameters which have an impact on the readability, and therefore understandability of code, as shown in .

Understanding the source code is impacted both by *legibility* (concerning syntax, and whether you can quickly visually scan the text and determine the main parts of the text, from blocks to words themselves) and *readability* (concerning semantics, whether you know the meaning of the words, and their role in the group) (Oliveira et al., 2020; Jacques & Kristensson, 2015). In 43 and 44, we show an excerpt of a function from the Tex-Live source code (Berry, 2022), formatted and unformatted.

IDEs therefore solve some of the mental operations performed by programmers when they engage with source code, such as representing code blocks through proper indentations. The automation of tooling and workflow increased in software such as Eclipse, IntelliJ, NetBeans, WebStorm Visual Studio Code³¹ has led to further entanglements of technology and appearance. By organizing code space through actions such as self documentation, folding code blocks, finding function declarations, batch reformatting and debug execution, they facilitate cognitive operations such as chunking, tracing, or highlighting beacons (Bragdon et al., 2010). These technical features show how a tool which operate at primarily the aesthetic level has consequences on the understandability of the system represented, even though this is, again, dependent on the skill level of the programmer (Kulkarni & Varma, 2017).

A significant dimension in which source code is being automatically formatted is the use of styleguides. The evolution of software engineering, from the individual programmer implementing ad hoc and personal so-

³¹Through which this thesis is written.

```

void texfile::prologue(bool deconstruct)
{
    if (inlinetex)
    {
        string prename = buildname(settings::outname(), "pre");
        std::ofstream *outpreamble = new
        → std::ofstream(prename.c_str());
        texpreamble(*outpreamble, processData().TeXpreamble, false,
        → false);
        outpreamble->close();
    }

    texdefines(*out, processData().TeXpreamble, false);
    double width = box.right - box.left;
    double height = box.top - box.bottom;
    if (!inlinetex)
    {
        if (settings::context(texengine))
        {
            *out << "\\definepapersize[asy][width=" << width <<
            → "bp,height="
            << height << "bp]" << newl
            << "\\setpapersize[asy][asy]" << newl;
        }
        else if (pdf)
        {
            if (width > 0)
                *out << "\\pdfpagewidth=" << width << "bp" << newl;
            *out << "\\ifx\\pdforigin\\undefined" << newl
                << "\\hoffset=-1in" << newl
                << "\\voffset=-1in" << newl;
            if (height > 0)
                *out << "\\pdfpageheight=" << height << "bp"
                    << newl;
            *out << "\\else" << newl
                << "\\pdforigin=0bp" << newl
                << "\\pdfvorigin=0bp" << newl;
            if (height > 0)
                *out << "\\pdfpageheight=" << height << "bp" << newl;
            *out << "\\fi" << newl;
        }
    }

    // ...
    if (!deconstruct)
        beginpage();
}

```

Listing 43: Example of a program text with syntax highlighting and machine-enforced indentation. See 44 for a functional equivalent, unformatted.

```

void texfile::prologue(bool deconstruct){if(inlinetex) {
    string prename=buildname(settings::outname(),"pre");
    std::ofstream *outpreamble=new std::ofstream(prename.c_str());
    texpreamble(*outpreamble,processData().TeXpreamble,false,false);
    outpreamble->close();
}

texdefines(*out,processData().TeXpreamble,false);
double width=box.right-box.left;
double height=box.top-box.bottom;
if(!inlinetex) {
    if(settings::context(texengine)) {
        *out << "\\definepaper[asy][width=" << width << "bp,height="
            << height << "bp]" << newl
            << "\\setuppaper[asy][asy]" << newl;
    } else if(pdf) {
        if(width > 0)
            *out << "\\pdffpagewidth=" << width << "bp" << newl;
        *out << "\\ifx\\pdfhorigin\\undefined" << newl
            << "\\hoffset=-1in" << newl
            << "\\voffset=-1in" << newl;
        if(height > 0)
            *out << "\\pdffpageheight=" << height << "bp"
                << newl;
        *out << "\\else" << newl
            << "\\pdfhorigin=0bp" << newl
            << "\\pdfvorigin=0bp" << newl;
        if(height > 0)
            *out << "\\pdffpageheight=" << height << "bp" << newl;
        *out << "\\fi" << newl;
    }
}
//...
if(!deconstruct)
beginpage();
}

```

Listing 44: Example of a program text without syntax highlighting nor machine-enforced indentation. See 43 for a functional equivalent, formatted.

lutions to a group of programmers coordinating across time and space to build and maintain large, distributed pieces of software, brought the necessity to harmonize and standardize how code is written—style guides started to be published to normalize the visual aspect of source code. These, called *linters*, are programs which analyzes the source code being written in order to flag suspicious writing (which could either be suspicious from a functional perspective, or from a stylistic perspective). They act as a sort of *intermediary object*, insofar as they assist individuals in the process of creating another object (Jeantet, 1998). Making use of formal syntax, IDEs' automatic styling of contributes to collective sense-making, something that we discuss further in 5.1.3.

This move from legibility (clear syntax) to readability (clear semantics) enables a certain kind of *fluency*, the process of building mental structures that disappear in the interpretation of the representations. The letters and words of a sentence are experienced as meaning rather than markings, the tennis racquet or keyboard becomes an extension of one's body, and so forth. Well-functioning interfaces are thus interfaces which disappear from the cognitive process of their user, allowing them to focus on ends, rather than on means (Galloway, 2012), leading to what Paul A. Fishwick has coined *aesthetic programming*, an approach of how attention paid to the representation of code in sensory ways results in better grasping of the metaphors at play in code.

Therefore, automatic tools operate at the surface-level but also with consequences at the deep-level, helping visualize and navigate the structure of a program text. In this case, we witness how computer-aided software engineering in the form of IDEs can be considered as a cognitive tool, a combination of surface representation affording direct interaction interface, whose formal arrangements and affordances facilitate direct engagement with the conceptual structures underlying in a program text. Perception and comprehension of source code is therefore more and more entangled with its automated representation.

Extended cognition

The roots of computer-enabled knowledge management can be found in the work of the encyclopedists, and scientists in seventeenth-century Europe, as they approached knowledge as something which could, and should be rationalized, organized and classified in order to be retrievable, comparable, and actionable (Sack, 2019). Scholars such as Roland Barthes, Jacques Derrida or Umberto Eco had specific knowledge-management techniques in order to let them focus on the arguments and ideas at hand, rather than on smaller organizational details, through the use of index cards; whether paper or digital, technology itself is a prosthesis for memory, an external storage which offloads the cognitive burden of having to remember things (Wilken, 2010).

Laying out his vision for a *Man-Computer Symbiosis*, J.C.R. Licklider, project leader of what would become the Internet and trained psychologist, emphasized information management. He saw the computer as a means to "*augment the human intellect by freeing it from mundane tasks*" (Licklider, 1960). By being able to delegate such mundane tasks, such as manually copying numbers from one document to another, one could therefore focus on the most cognition-intensive tasks at hand. While improving input, speed and memory of contemporary hardware has supported Licklider's perspective a single limitation that he pointed out in the 1950s nonetheless remains: the problem of language.

What we want to accomplish, and how do we want to accomplish it, are complex questions for a computer to process. The subtleties of language imply some ambiguities which are not the preferred mode of working of a logical arithmetic machine. If machines can help us think, there are however some aspects of that thinking which cannot easily be translated in the computer's native, formal terms, and the work of interface designers and tool constructors has therefore attempted to automate most of what can be automated away, and facilitate the more mundane tasks done

a by a programmer. Software tools are therefore used to think and explore concepts, by supporting epistemic actions in various modalities (Victor, 2014).

The computer therefore supports epistemic actions through its use of metaphors (to establish a fundamental base of knowledge) and of actions (to probe and refine the validity of those metaphors) to build a mental model of the problem domain. In the case of IDEs, the problem domain is the source code, and these interfaces, by allowing means of scanning and navigating the source code, are part of what Simon Penny calls, after Clark and Chalmers, *extended cognition* (Penny, 2019). Extended cognition posits that our thinking happens not only in our brains, but is also located in the tools we use to investigate reality and to deduce a conceptual model of this reality based on empirical results. We consider IDEs a specific manifestation of embodied cognition, actively helping the programmer to define, reason about, and explore a code base. The means of taking epistemic action, then, are also factors in contributing to our understanding of the program text at hand. In this spirit, David Rokeby goes as far as qualifying the computer as a *prosthetic organ for philosophy*, insofar as it helps him formulate accurate mental models as he interacts with them through computer interfaces, compensating for its formal limitations³².

This brings us back to our discussion of Simondon's technical and aesthetic modes of existence 3.2.1. As highlighted by the use of software tools in the sense-making process of a program text, formal syntax only operates on distinct, fragmented concepts, as evoked in the technological mode³³.

³²"The fact that words can be stored and manipulated by a computer does not mean that the referenced concepts or material reality are held in the computer. We reinvigorate a computer's textual output with our mind's wet and messy renderers. The computer is just holding on to given patterns, sets of unambiguous measurements of key-strokes, mouse-clicks, modem songs, sensor reading..." (Rokeby, 2003)

³³Rokeby further develops on the computer's fragmentation process, which he calls quantification: "The material world cannot enter into this digital nirvana except through that particular "eye of the needle" called quantification, that most literal and unforgiving form of en-

In turn, the aesthetic mode, expressed through the more systemic and totalling approach of metaphors and of sensual perception, can compensate this fragmenting process. This does suggest that the cognitive process of understanding technical artifacts, such as source code, necessitates complementary technical and aesthetic modes of perception.

Programmers face the complexity of software on a daily basis, and therefore use specific cognitive tools to help them. While our overall argument here is that aesthetics is one of those cognitive tools, we focused on this section on two different, yet widely used kinds: the metaphor and the integrated development environment.

We pointed out the role that metaphors play in creating connections between pre-existing knowledge and current knowledge, building connections between both in order to facilitate the construction of mental models of the target domain. Metaphors are used by programmers at a different level, helping them grasp concepts (e.g. memory, objects, package) without having to bother with details. As we will see in the following chapters (see 4.2.1 and 5.2.3), metaphors are also used by programmers in the source code they write in order to elicit this ease of comprehension for their readers.

Programmers also rely on specific software tools, in order to facilitate the scanning and exploring of source code files, while running mundane tasks which should not require particular programmer attention, such as linking or refactoring. The use of software to understand software is indeed paradoxical, but nonetheless participates in extended cognition; the means which we use to reason about problems affect, to a certain extent, the quality of this reasoning.

coding." (Rokeby, 2003)

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)

This chapter has shown that software is a complex object, an *abstract artifact*, existing at multiple levels, and in multiple dimensions. Programmers therefore need to deal with this complexity and deploy multiple techniques to do so. Psychology studies, investigating how programmers think, have pointed out several interesting findings. First, building mental models from reading and understanding source code is not an activity which relies exclusively on the part of the brain which reads natural language, nor on the part which does mathematical operations. Second, the reasoning style is multimodal, yet spatial, involving layered abstractions; programmers report working and thinking at multiple levels of scale, represent parts of code as existing closer or further from one another, in non-linear space. Third, the form affects the content. That is, the way that code is spatially and typographically laid out helps, to a certain, with the understanding of said code, without affecting expertise levels, or guaranteeing success.

In order to deal with this complexity, some of the means deployed to understand and grasp computers and computational processes are both linguistic and technical. Linguistic, because computer usage is riddled with metaphors which facilitate the grasping of what the presented entities are and do. These metaphors do not only focus on the end-users, but are also used by programmers themselves. Technical, because the writing and reading of code has relied historically more and more on tools, such as programming languages and IDEs, which allows programmers to perform seamless tasks specific to source code.

In the next chapter, we pursue our inquiry of the means of understand-

ing, moving away from software, and focusing on how the aesthetic domains examined in 2.2. This will allow us to show how source code aesthetics, as highlighted by the metaphorical domains that refer to it, have the function of making program texts understandable.

Chapter 4

Beauty and understanding

This chapter provides background argumentation for what beauty has to do with understanding. First from a theoretical perspective, and then diving specifically into how specific domains approach this relation. Our theoretical approach will start from the aesthetic theory of Nelson Goodman, and a lineage which links aesthetics to cognition, most recently aided by the contribution of neurosciences. We will see how source codes does qualify as a language of art—that is, a symbol system which allows for aesthetic experiences.

After arguing for a conception of aesthetics which tends to intellectual engagement, we will pay attention to how surface structure and conceptual assemblages relate. That is, we will highlight how each of the domains contingent to source code—literature, mathematics and architecture—communicate certain concepts through their respective and specific means of symbolic representation. The identification of how specific aesthetic properties enable cognitive engagement in each of these domains will in turn support the identification of how equivalent properties can manifest in source code.

This thesis argues that aesthetics have a useful component, insofar as formal arrangements at the surface-level can facilitate the understanding of

the underlying deep structure of concepts denoted. In the specific context of source code, we show that aesthetic standards are contextual, as they vary along two axes. First, they depend on whether the attention of the writer (and thus the reader) is directed at the hardware, or at the software (which can, in turn, address real-world ideas, or computational ideas). Second, they depend on the socio-technical context in which source code is written, a context constituted of whether the program text is read-only or read-write, and of whether the intent is for the program text to be primarily functional, educational or entertaining.

4.1 Aesthetics and cognition

The way that things are presented formally has been empirically shown to affect the comprehension of content. Without engaging too directly in the media-determination thesis, which states that what one can say is determined by the medium through which they say it, be it language or technical media (Postman, 1985), we nonetheless do start from the point that form influences the perception of content.

Jack Goody and Walter Ong have shown in their anthropological studies that the primary means of communication of the surveyed communities does affect the engagement of said communities with concepts such as ownership, history and governance (Ong, 2012; Goody, 1986). More recently, Edward Tufte and his work on data visualization have furthered this line of research by focusing on the translation of similar data from textual medium to graphic medium (Tufte, 2001). Several cases have thus been made for the impact of appearance towards structure, both in source code and elsewhere. Here, we intend to generalize this comparative approach between several mediums, by looking at how source code performs expressively as a language of art, stemming from Nelson Goodman's theorization of such a languages.

4.1.1 Source code as a language of art

Moving away from the question of the nature of the aesthetic experience from the perspective of the audience, whether as an aesthetic emotion being felt or as an aesthetic judgment being given, we shift our attention to the object of aesthetic experience, and to the questions of *how does a program text represent?* and *what does a program text represent?*. To answer these, we rely on the approaches provided by Nelson Goodman in the *Languages of Art: An Approach to a Theory of Symbols* (Goodman, 1976).

The starting point for Goodman's analysis is that production and understanding in the arts involve human activities that, though they differ in specific ways among themselves and from other activities, are nevertheless generically related to perception, scientific inquiry, and other cognitive activities, since both artistic and scientific activities involve symbolic systems. It is those two components that Goodman aims at expliciting: what constitutes an aesthetic symbol system, and how does such a system express?

Goodman develops a systematic approach to symbols in art, freed from any media-specificity (e.g. f from clocks to counters, from diagrams to maps models, from musical scores to painters' sketches and linguistic scripts). A symbolic system, in his definition, consists of characters, along with rules to govern their combination with other characters, itself correlated with a field of reference. These symbols and their arrangement within a work of art supports an aesthetic experience¹ and, since they are syntactic system which operate at the semantic level, they can be rigorous communicative systems.

A symbol system is based on requirements which might indicate that the work created in such a system would be able to elicit an aesthetic ex-

¹It should be noted here that Goodman does not limit the aesthetic experience to a positive, pleasurable one. An artistic symbolic system can be seen even if the result is considered bad.

perience². Such a system should be composed of signs which are syntactically and semantically disjointed, syntactically replete and semantically dense (Goodman, 1976). This classification makes it possible to compare the way various symbolization systems used in art and science express concepts. In our case, this provides us for a framework to investigate the extent to which source code qualifies as a language of art.

Source code is written in a formal linguistic system called a programming language. Such a linguistic system is digital in nature, and therefore satisfies at least the two requirements of syntactic disjointedness (no mark can be mistaken for another) and differentiation (a mark only ever corresponds to that symbol). Indeed, this is due to the fact that these requirements are fulfilled by any numerical or alphabetical system, as programming languages are systems in which alphabetical characters are ultimately translated into numbers. While not as syntactically dense as music or paint, it is nonetheless unambiguous.

Third, the requirement of syntactic repleteness demands that relatively fewer factors need to be taken into account during the interpretative process³. On one hand, we can consider that any additional aspects of the source code (such as the display font or the syntax highlighting discussed in 3.3.2) are ultimate irrelevant to the computer, thus making it a poorly replete symbol system. On the other hand, the importance of such factors, along with abilities to write a program with the same function but with different syntax, pleads for a relatively replete syntactical system. The tendency of program text to veer towards verbosity indeed implies this desir-

²Goodman approaches it as such: "*Perhaps we should begin by examining the aesthetic relevance of the major characteristics of the several symbol processes involved in experience, and look for aspects or symptoms, rather than for crisp criterion of the aesthetic. A symptom is neither a necessary nor a sufficient condition for, but merely tends in conjunction with other such symptoms to be present in, aesthetic experience*" (Goodman, 1976)

³Goodman mentions the symptom that such a system might engender: "[...] relative syntactic repleteness in a syntactically dense system demands such effort at discrimination along, so to speak, more dimensions" (Goodman, 1976)

able state of repleteness: more subtleties and intermediate syntax can be added within any proposition, always implying the possibility of clarifying, or obfuscating—both being, as we have seen, different kinds of aesthetic experiences.

Finally, semantic density refers to whether or not there is a limit to the amount of concepts that the symbol system can refer to. As we have shown in 4.5, the affordances that programming languages provide to represent phenomena and concepts from the problem domain fulfill this requirement. While we have been previously concerned with syntax, this ability of programming languages to refer to a problem domain which has not yet shown its limitations at the semantic level is one which gives it representational power beyond strict computational concepts.

As Goodman notes, the distinct signs that compose a symbols system do not have intrinsic properties, but a mark serves as a sign only in relation to a symbol system, and to a field of reference. The field of reference is understood here as being the set of concepts which are being referred to by a symbolic system. For instance, a symbolic system such as western classical music can refer to concepts such as lament, piety, heroism or grace, while a chinese *shanshui* painting has a landscape composed of mountains and rivers, as well as concepts of harmony, complementarity, presence and absence, as its field of reference. The combination of both the problem domain, as evoked in 3.2.2, and of the technological environment on which the source code is to be executed, developed in 5.1.3, are posited here as an equivalent to the Goodman's field of reference.

It thus seems like source code satisfies to a large extent the critieria to be a language of art, meaning that it exhibits some of the properties which tend to elicit an aesthetic feeling. Most notably, it does not possess a very dense syntax, nor can it be considered replete both from the perspective of the computer and of the human⁴, but it nonetheless refers possesses a

⁴See 5.1.1 for a discussion of syntactic limitation in programming languages, also known as orthogonality.

certain amount of semantic density. Its ability to connect to a particular field of reference, such as hardware, mathematics, or the world at large is another aspect of being a language of art, and is an important part of how programming languages can communicate concepts.

Goodman highlights the ways in which symbols systems communicate, through the notion of *reference*. To refer to, in this sense, is the action by which a symbol stands in for an item or an idea. Reference, he sketches out, takes place through the different dyads of denotation and exemplification, description and representation, possession and expression (Goodman, 1976). We will see how these various means of referring can be instantiated in the symbolic system of source code.

Denotation is the core of representation, a reference from a symbol to one or many objects it applies to and is independent of resemblance. To refer, it uses a particular relationship via the use of labels, in which a symbol stands in for an item in the field of reference. For instance, a name denotes its bearer and a predicate each object in its extension. Names such as variable names or function names thus denote a particular item in the field of reference, and act as their label. For instance, `var auth_level` denotes an ability to access and modify resources; the first token `var` is chosen by the language designer, while the second token `auth_level` is chosen by the programmer.

The labelling process therefore serves as the symbolic expression for a particular field. In source code, this can happen through variable naming, but also through type definition⁵, as well as additional affordances which we look at in 5.2, such as the layering of semantic references and the establishment of habitable cognitive structures.

Source code also make extensive use of description. If we consider a program text as a series of steps, a series of states, or a series of instructions, then it follows that source code is explicitly describing the algorithm

⁵For instance, a particular choice of a numeric value, such as `int` or `float` denote a particular level of precisionness

```

class Person {
    int age;
    String name;
    Interest[] interests;

    void greet(){
        System.out.println("hi, my name is "+name+"!")
    }
}

class Interest {
    int priority;
    String name;
}

```

Listing 45: An example of how source code can be a representation an individual, and can exemplify encapsulation, written in Java.

used—the how of the program, rather than the why. Indeed, a program text is a description of how to solve a problem from the computer’s perspective, written extensively in machine language⁶. All source code can therefore be said to be a description of a combination of states (data) and actions (functionality).

States are also a particular case in source code: they are both a description and, because they are not the thing itself, they are also a representation. As one can see in 45, an individual can be represented within source code with a particular construct in which states and actions are encapsulated. Interestingly, this representation of a concept as an object in source code does not imply that it reveals the intrinsic properties of the object; rather, these properties appear as they are given by the modelling process of source code syntax. As a symbol system, source code thus proposes a model of the world in which objects have properties; a slightly different representation is therefore always possible.

This representation, in the specific instance of object-oriented programming in 45, also manifests Goodman’s aesthetic symptom of posses-

⁶Pseudo-code is therefore a representation of a potential source code written in a specific language.

sion. Here, the source code possesses similar properties as the thing referenced (since our prototypal image of a person has an age, a name and interests). Through this possession of a property, it acts as an example of a prototypal person.

Exemplification is another aspect of Goodman's theory, which has nonetheless remained somewhat limited (Elgin, 2011). A symbol exemplifying, also called an exemplar, is considered as a stand-in for an item in the field of reference. We have seen source code act as an example in 2.1.3, where a particular program text is written in order to stand in for a broader concept. For instance, a program text can, at a lower level, exemplify a particular kind of procedure, such as encapsulation (see 45) or nestedness. The program text therefore exemplifies the constitutive element of the linked list⁷. However, a similar program text can also be an example of cleanliness, of clarity, or elegance. A program text written by a software developer can be seen as possessing the property of cleanliness (see 28 in 2.2.2), by virtue of its implementation of syntactic and semantic rules, while another program text written by a hacker can be seen as highlighting detailed hardware knowledge (see 30 in 2.2.2).

Different implementations of a concept are necessary but not sufficient for aesthetic judgment, whether these different implementations are virtual or actual. The comparative approach is the one which enables the labelling of *good* or *bad* only insofar as there is a relative *worse* or *better*, respectively. Additionally, the features which a symbol exemplifies always depend on its function (or, more precisely, its functional context) (Elgin, 1993). As we show in 5.3.2, a symbol can perform a variety of functions: a piece of code in a textbook might exemplify an algorithm, while the same piece of code in production software might be seen as a liability, or denote boredom in a code poem. It is then both the possibility of alternative implementations and the reality of the current implementation context

⁷A linked list is a basic data structure in computer science, which consists in a succession of connected objects.

which give the exemplification of program texts its aesthetic potential.

Source code maintains a specific kind of relation to the field of reference. The particular class of characters employed as symbols (called *tokens* in the context of programming languages), involves a separation between name, value and address, and as such does not guarantee a direct relationship with the items in the field of reference, we can see in the line `unsigned three = 1;` of 46, where the reference of the name is not the same reference as the value. That is, in program texts, two distinct symbols can be referring to the same concept, value, or place in memory, something Goodman nonetheless assigns as another symptom of the aesthetic: multiple and complex references.

On the other hand, the representation of a field of reference is done through a disjointed and differentiated system: the boundaries of each items in the field of reference are clearly defined, in virtue of the specific symbol system that programming languages are. It is their combination which, in turn, enables complex interplay of references.

We have shown here that source code qualifies as a symbolic system susceptible of affording symptoms of the aesthetic. We have also highlighted its specificities, particularly in terms of descriptions and representations through a restricted syntactic system enabling complex and multiple references, due to it being a language across human and machine understanding. Source code is thus written in a specific kind of symbol system, one which counts as a language of art, but does with restricted syntax and expansive semantics.

A final aspect to investigate is the expressiveness of source code, with a particular attention to how source code can manifest of metaphorical exemplification and representation. One particular expressive power of an aesthetic experience surfaces when the exemplification involves a foreign element, an event that Goodman refers to as metaphorical exemplification. While this approach has been broadened by Lakoff et. al., and mentioned in 3.3.1, other philosophers of art have also pinpointed the metaphorical

```

static int verify_reserved_gdb(struct super_block *sb,
                               ext4_group_t end,
                               struct buffer_head *primary)
{
    const ext4_fsblk_t blk = primary->b_blocknr;
    unsigned three = 1;
    unsigned five = 5;
    unsigned seven = 7;
    unsigned grp;
    __le32 *p = (__le32 *)primary->b_data;
    int gbackups = 0;

    while ((grp = ext4_list_backups(sb, &three, &five, &seven)) < end)
    {
        if (le32_to_cpu(*p++) != grp * EXT4_BLOCKS_PER_GROUP(sb) + blk)
        {
            ext4_warning(sb, "reserved GDT %llu"
                        " missing grp %d (%llu)",
                        blk, grp,
                        grp *
                        (ext4_fsblk_t)EXT4_BLOCKS_PER_GROUP(s_
                            ↪ b)
                            ↪ +
                            ↪
                            blk);
            return -EINVAL;
        }

        if (++gbackups > EXT4_ADDR_PER_BLOCK(sb))
            return -EFBIG;
    }
    return gbackups;
}

```

Listing 46: An example from the Linux kernel showing that the name and the value of a variable might refer to different things (Linux, 2023).

event as a reliable symptom of the aesthetic.

Max Black initiates a view of metaphors which go beyond a simple comparison; dubbed the *interaction view*, he considers the metaphorical device as containing positive cognitive content, rather than simply entertaining or limiting (Black, 1955). Against a traditional view of metaphor being a word which stands in for another, Black reveals a large web of interactions which prove harder to disentangle, beyond usual similarities between two words⁸. Simply paraphrasing a metaphor, even if one captures precisely the same connotations/associations as the metaphor, does not convey the same meaning as the metaphor itself. For instance, saying "*Je chavire dans l'embrun des phénomènes*"⁹ (Beckett, 1982) does not have the similar expressive power as listing all the properties of *phénomènes*. The use of the verb capsize in conjunction with spray relates to the domain of navigation, while capsize alone tends more to a dynamic movement, and spray to uncertainty and bluriness of shape. Phenomenas of the world are all requalified in the light of these new kinetic and perceptual associations.

Through his contribution to aesthetic philosophy, Monroe Beardsley's started touching upon metaphor from a semantic perspective. Published alongside his inquiries into the aesthetic character of an experience, *The Metaphorical Twist* implies that semantics and aesthetics might be connected through the structuring operation of the metaphor—that which elicits an aesthetic experience can do so through the creation of unexpected, or previously unattainable meaning. Beardsley's conception is that metaphor can have a designative role (the primary subject) which adds a

⁸"Reference to 'associated commonplaces' will fit the commonest cases where the author simply plays upon the stock of common knowledge (and common misinformation) presumably shared by the reader and himself. But in a poem, or a piece of sustained prose, the writer can establish a novel pattern of implications for the literal uses of the key expressions, prior to using them as vehicles for his metaphors. [...] Metaphors can be supported by specially constructed systems of implications, as well as by accepted commonplaces; they can be made to measure and need not be reach-me-downs." (Black, 1955).

⁹Literally translated as "I capsize in the spray of phenomena"

"local texture of irrelevance", a *"foreign component"*, whose semantic richness might over-reach and obfuscate the intended meaning, as well as a connotative one (the secondary subject), in which meaning is peripheral (Beardsley, 1962). The cognitive stimulation and enlightenment takes place through a metaphor-induced tension, between central and periphery, between illuminating and obfuscating, between evidence and irrelevance.

As Beardsley inquiries into the features necessary for an aesthetic experience, of which the metaphor is part, he lists five criteria to distinguish the character of such an experience. Besides object-directedness, felt-freedom, detached-affect and wholeness, is the criteria of *active discovery*, which is

a sense of actively exercising the constructive powers of the mind, of being challenged by a variety of potentially conflicting stimuli to try and make them cohere; exhilaration in seeing connections between percepts and meanings; a sense of intelligibility (Beardsley, 1970).

As such, Beardsley highlights the possibility of an aesthetic experience to make understandable, to unlock new knowledge in the beholder, and he considers metaphors as a way to do so. The stages he lists go from (1) the word exhibiting properties, to (2) those properties being made into meaning, and finally into (3) a staple of the object, consolidating into (or dying from becoming) a commonplace. This interplay of a metaphor being integrated into our everyday mental structures, of poetry bringing forth into the thinkable, and in the creation of a tension for such bringing-forth to happen, makes the case for at least one of the consequences of an aesthetic experience, and therefore one of its functions: making sense of the complex concepts of world.

Finally, Catherine Elgin has pursued the work of Goodman by furthering the inquiry into arts as a branch of epistemology. Drawing on the work mentioned above, she investigates the relationship between art and understanding, considering how interpretively indeterminate symbols advance

understanding (Elgin, 2020), and that it does so in the context of interpretive indeterminacy. As syntactically and semantically dense symbol systems are used in artworks, it is this multiplicity in interpretations which requires sustained cognitive attention with the artwork. To explain these multiple interpretations, the metaphor is again presented the key device in explaining the epistemic potency of aesthetics, based on an interpretative feedback loop from the viewer. And yet, in the context of source code, this interpretation is always shadowed by its machine counterpart—how the computer interprets the program.

4.1.2 Contemporary approaches to art and cognition

We have drawn from existing work in philosophy of art, in order to map out the expressive power of a given formal representation, as a traditional prerequisite to the gaining of art status of an object, and highlighted the role of metaphors in engaged cognition during an aesthetic experience. Contemporary literature, and the emergence of neuroscientific studies of such aesthetic experience seem to confirm empirically this approach, and highlight as well two related additional components: sequential experience and skill levels.

The aesthetic experience—that is, the positively received perception of a natural or crafted object—has traditionally been laid out across multiple axes, with more or less overlap. The axes involved in this positive perception include an emotional response, a harmonious assessment, an axiomatic adherence or disinterested pleasure, and have been the topic of debates amongst philosophers for centuries (Peacocke, 2023).

Noël Carroll sums up these different directions under the broad areas of affect, axiom and content ultimately considering a content-based approach as the most fruitful (Carroll, 2002). First, he underlines how an aesthetic experience dictated by affect removes the object from one's assessment of purpose, value and effect, and limiting it to form, following

Kant's principle of disinterested pleasure via passive contemplation. As such, a flower, a sunset or a musical melody can evoke affective aesthetic experiences. Yet, the supposed tendency of this kind of experience to release us from worldly concerns fails, for Carroll, to encompass aesthetic experiences that are rooted in so-called worldly concerns—such as a documentary photography, skillful physical performance, or delicately crafted glassware—and is therefore unsatisfying as a root explanation for the aesthetic experience.

An axiomatic aesthetic experience is based on the sort of value that the object is being associated with—such as depiction of religious topics or a manifestation of a particular style. While Carroll does acknowledge a certain virtue of this aesthetic experience in terms of contribution to group cohesion through shared values and imaginaries, its limitations are found in a pre-existing answer to the value judgment that is being bestowed upon the object: the material and sensual properties of the object at hand are irrelevant since their quality is already decided *a priori*.

It is in the content approach that Carroll finds the most satisfying answer to what the aesthetic experience is. Content, here, is defined as the significant forms being apprehended, along with its combinations, juxtapositions and comparisons with other forms¹⁰. When we engage with the sensual aspects of an object, our attention is indeed directed first and foremost at what the object looks like, rather than how it makes one feel, or what value system it belongs to. More specifically, Carroll notes, if attention is directed with understanding to the form of the art work or to its expressive and aesthetic properties or to the interaction between those features, then the experience is said to be aesthetic (Carroll, 2002).

Form, and the attention paid to it, will thus be taken as our starting

¹⁰"Whereas affect-oriented approaches tend to identify aesthetic experience in terms of certain distinctive experiential qualia or feeling tones, such as being lifted out of the flow of life, content-oriented approaches proceed by distinguishing the specific objects of said experiences." (Carroll, 2002).

point. This content approach to form, i.e. the set of appearing choices intended to realize the purpose of the artwork, also involves questions of function, implied by the presence of purpose pertaining to an artwork. Particularly, how does the object of aesthetic experience manifest such a purpose, in a way that it can be correctly judged, insofar as its perceived form and perceived purpose are aligned, distinct from any emotional or axiomatic charge?

We can find an answer in the study conducted by Anjan Chatterjee and Oshin Vartanian on the evaluation of the aesthetic experience from a neuroscientific point of view. Like Carroll, they highlight three different perspectives: a sensory-motor perspective, loosely mapped to an affective experience, an emotion-valuation perspective, similar to an axiological experience, and a meaning-knowledge experience, which we equate to the content approach to the aesthetic experience (Chatterjee & Vartanian, 2016).

Importantly, they make the distinction between an aesthetic judgment, which emanates from the process of understanding the work, and an aesthetic emotion, which follows from the ease of acquisition of such an understanding. Without being mutually exclusive, these two pendants are related to the amount of engagement provided by the person who aesthetically experiences the object. One can have an aesthetic emotion without being able to provide an aesthetic judgment, a case in which one does not hold enough expertise to apprehend or appreciate a particular realisation. In this sense, the aesthetic judgment, unlike the aesthetic emotion, requires something additional. This conditioning of the aesthetic experience to a certain kind of pre-existing knowledge or skill is supported by the authors' mention of the theory of fluency-based aesthetics (Chatterjee & Vartanian, 2016), and their view builds on models that frame aesthetic experiences as the products of sequential and distinct information-processing stages, each of which isolates and analyzes a specific component of a stimulus (e.g., artwork).

These stages, drawn from Leder et. al's model, are based on empirical

observation in scientific studies which segment an aesthetic experience in sequential steps (Leder et al., 2004). These evolve from perception, to implicit classification, explicit classification, cognitive mastering and finally evaluation—that is, fully-qualified aesthetic judgment. This conception is concomitant to Reber et. al.'s proposal for an aesthetic framework based on processing fluency, which they define as a function of the perceiver's processing dynamics: the more fluently the perceiver can process an object, the more positive is her aesthetic response (Reber et al., 2004). While they focus their study on perceptual fluency, tending to traditional aesthetic features such as symmetry, contrast and balance; they also consider conceptual fluency as an influence on the aesthetic experience, through the attention given to the meaning of a stimulus and the relation of form to semantic knowledge structures. Such a conceptualizing thus hints at a similar skill-based, contextual framework which we have seen emerge in the aesthetic judgment of source code, and yet an additional establishment of a relation between truth and beauty¹¹.

This approach of cognitive ease, which we've already identified in the conclusion of 2, is finally echoed in the view that Gregory Chaitin, a computer scientist and mathematician, offers of comprehension as compression. By considering that the understanding of a topic is correlated with the lower cognitive burden experienced when reasoning about such topic, Chaitin forms a view in which an individual understands better through a properly tuned model—a model that can explain more with less (Zenil, 2021). In this sense, aesthetics help compress concepts, which in turn allows someone to hold more of these concepts in short-term memory, and grasp a fuller picture, so to speak.

These studies thus show a particular empirical attention to the cogni-

¹¹"these findings suggest that judgments of beauty and intuitive judgments of truth may share a common underlying mechanism. Although human reason conceptually separates beauty and truth, the very same experience of processing fluency may serve as a nonanalytic basis for both judgments." (Reber et al., 2004)

tive engagement with respect to the apprehension of an object from an aesthetic perspective, as opposed to passive contemplation or value-driven agreement. While these other types of experiences remain valid when apprehending such an object, we do focus here on this specific kind of experience: the cognitive approach to the aesthetic experience. Going back to Goodman, he describes such an experience as involving:

making delicate discriminations and discerning subtle relationships, identifying symbol systems and what these characters denote and exemplify, interpreting works and reorganizing the world in terms of works of art and works in terms of the world.
(Goodman, 1976)

In this section we've glanced at an overview of research on how cognitive engagement is involved in an aesthetic experience, both from the point of view of the philosophy of art and from cognitive psychology. However, highlighting this involvement does not immediately explicit the nature and details of such cognitive engagement. Speaking in terms of form and object are higher-level concepts tend to erase the specificities of the various systems of aesthetic properties, and how their arrangement expresses various concepts.

Now that we have sketched out an understanding of source code as a symbolic system supporting an aesthetic experience, we must provide a more detailed account of the specificities of source code. To do so, we first turn to a comparative approach, looking at the set of aesthetic domains metaphorically connected to source code through programmer discourse, and we analyse how each of these domains involve cognition in their formal presentations.

4.2 Literature and understanding

Literature as a cognitive device relies, as we've seen in 2.2, on the use of metaphors to provide a new perspective on a familiar concept, and hence complement and enrich the understanding that one has of it. While Lakoff and Johnson's approach to the conceptual metaphor will serve a basis to explore metaphors in the broad sense across software and narrative, we also argue that Ricoeur's focus on the tension of the *statement* rather than primarily on the *word* will help us better understand some of the aesthetic manifestations of software metaphors, without being limited to tokens, but going beyond to statement and structure. Following a brief overview of his contribution, we examine the various uses of metaphor in software and in literature, touch upon the cognitive turn in literary studies, and conclude with an account of how this turn involves further thinking into the spatial and temporal properties of the written word.

4.2.1 Literary metaphors

Writing in *The Rule of Metaphor*, Ricoeur operates two shifts which will help us better assess not just the inherent complexity of program texts, but the ambivalence of programming languages as well. His first shift regards the locus of the metaphor, which he saw as being limited to the single word—a semiotic element—to the whole sentence—a semantic element (Ricoeur, 2003). This operates in parallel with his attention to the *lived* feature of the metaphor, insofar it exists in a broader, vital, experienced context. Approaching the metaphor while limiting it to words is counter-productive because words refer back to "contextually missing parts"—they are eminently overdetermined, polysemic, and belong to a wider network meaning than a single, Aristotelian, one-to-one relationship. Looking at it from the perspective of the sentence brings this rich network of potential meanings and broadens the scope for and the depth of interpretation.

As we develop in 5.2.3 in our reading of 77, not all of the evocative meaning of the poem are contained exclusively in each token, and the power of the whole is greater than the sum of its parts.

Secondly, Ricoeur inspects a defining aspect of a metaphor by the *tensions* it creates. His analysis builds from the polarities he identifies in discourse between event (time-bound) and meaning (timeless), between individual (subjective, located) and universal (applicable to all) and between sense (definite) and reference (indefinite). The creative power of the metaphor is its ability to both create and resolve these tensions, to maintain a balance between a literal interpretation, and a metaphorical one—between the immediate and the potential, so to speak. Tying it to the need for language to be fully realized in the lived experience, he poses metaphor as a means to creatively redescribe reality. In the context of syntax and semantics in programming languages, we will see that these tensions can be a fertile ground for poetic creation through aesthetic manifestations. For instance, we can see in 47 a poetic metaphor hinging on the concept of the attribute. In programming as in reality, an attribute is a specificity possessed by an entity; in this code poem, the tension is established between the computer interpretation and the human interpretation of an attribute. Starting from a political target domain (the constitution of the United States of America), the twist happens in the source domain of the attribute. Loosely attributed by the people in writing, the execution of the declaration (that is, the living together of the United States citizens) implies and relies on the fact that power resides in the people, as is being stated in a literal way. However, from the computer perspective, the definition is not rigorous enough and the execution of the code will throw an error that is shown on the last line—the people have no power.

In such case, the expressiveness of the program text can be said to derive from the continuous threading of metaphorical references, weaving the properties of computational objects and the properties of conceptual objects in order to deep the mapping from one unto the other.

```

title = 'Constitution of the United States'

preamble = { 'Preamble': "We the People of the United States, \
in Order to form a more perfect Union, \
establish Justice, insure domestic Tranquility, \
provide for the common defense, promote the general Welfare, \
and secure the Blessings of Liberty to ourselves and our Posterity, \
do ordain and establish this Constitution for the United States of
↪ America." }

WEPOTUS_power = { 'ordain_and_establish' : lambda x, y :
↪ Constitution(x, y )}

WEPOTUS = People("We the People of the United States", WEPOTUS_power)

WEPOTUS.GOALS = [ "form a more perfect Union",
"establish Justice",
"insure domestic Tranquility",
"provide for the common defense",
"promote the general Welfare",
"secure the Blessings of Liberty to ourselves and our Posterity"
]

USConstitution = WEPOTUS.power['ordain_and_establish'](title,
↪ preamble)

AttributeError: 'People' object has no attribute 'power'

```

Listing 47: Cynical American Preamble, by Michael Carlisle, published in code::art #o (Brand, 2019)

So while Lakoff bases poetic metaphors on the broader metaphors of the everyday life, he also operates the distinction that, contrary to conventional metaphors which are so widely accepted that they go unnoticed, the poetic metaphor is non-obvious. Which is not to say that it is convoluted, but rather that it is new, unexpected, that it brings something previously not thought of into the company of broad, conventional metaphors—concepts we can all relate to because of the conceptual structures we are already carry with us, or are able to easily integrate.

Poetic metaphors deploy their expressive powers along four different axes, in terms of how the source domain affects the target domain that is connected to. First, a source domain can *extend* its target counterpart: it pushes it in an already expected direction, but does so even further, sometimes creating a dramatic effect by this movement from conventional to poetic. For instance, a conventional metaphor would be saying that "*Juliet is radiant*", while a poetic one might extend the attribution of positivity and dramatic important associated with brightness and daylight by saying "*Juliet is the sun*"¹².

Poetic metaphors can also *elaborate*, by adding more dimensions to the target domain, while nonetheless being related to its original dimension. Here, dimensions are themselves categories within which the target domain usually falls (e.g. the sun has an astral dimension, and a sensual dimension). Naming oneself as *The Sun-King* brings forth the additional dimension of hierarchy, along with a specific role within that hierarchy—the sun being at the center of the then-known universe.

Metaphors gain poetic value when they *put into question* the conventional approaches of reasoning about, and with, a certain target domain. Here is perhaps the most obvious manifestation of the *non-obvious* requirement, since it quite literally proposes something that is unexpected from a conventional standpoint. When Albert Camus describes

¹²From *Romeo and Juliet*, Act 2, Scene 2

Tipasa's countryside as being *blackened from the sun*¹³, it subverts our pre-conceptions about what the countryside is, what the sun does, and hints at a semantic depth which would go on to support a whole philosophical thought, knowns as *la pensée de midi*, or *the noon-thought*¹⁴.

Finally, poetic metaphors *compose* multiple metaphors into one, drawing from different source domains in order to extend, elaborate, or question the original understanding of the target domain. Such a technique of superimposition creates semantic depth by layering these different approaches. It is particularly at this point that literary criticism and hermeneutics appear to be necessary to expose some of the threads pointed out by this process. As an example, the symbol of Charles Bovary's cap, a drawn-out metaphor in Flaubert's *Madame Bovary* ends up depicting something which clearly is less of a garment and more of an absurd structure, operates by extending the literal understanding of how a cap is constructed, elaborating on the different components of a hat in such a rich and lush manner that it leads the reader to question whether we are still talking about a hat (Nabokov, 1980). This metaphorical composition can be interpreted as standing for the orientalist stance which Flaubert takes vis-à-vis his protagonists, or for the absurdity of material pursuit and ornament, one which ultimately leads the novel's main character, Emma, to her demise, or for the novel itself, whose structure is composed of complex layers, under the guise of banal appearances. Composed metaphors highlight how they exist along *degrees of meanings*, from the conventional and expected to the poetic and enlightening.

We have therefore highlighted how metaphors *function*, and how they

¹³"A certaines heures, la campagne est noire de soleil." (Camus, 1972)

¹⁴Interestingly, the re-edition of L'Étranger for its 70th anniversary can itself be seen as a form of poetic metaphor, since it was published under Gallimard's *Futuropolis* collection. While the actual *Futuropolis* doesn't claim to focus on any sort of science-fiction publications, and rather on illustrations, the very name of the collection applies onto the work of Camus, and of the others published alongside him, can elicit in the reader a sense of a kind of avant-gardism that is still present today.

can be identified. Another issue they address is that of the *role* they fulfill in our everyday experiences as well as in our aesthetic experiences. Granted a propensity to structure, to adapt, to reason and to induce value judgment, metaphors can ultimately be seen as a means to comprehend the world. By importing structure from the source domain, the metaphor in turn creates cognitive structure in the target domains which compose our lives. Our understanding grasps these structures through their features and attributes, and integrates them as a given, a reified convention—in what Ricoeur would call a *dead* metaphor. This is one of their key contribution: metaphors have a function which goes beyond an exclusive, disinterested, self-referential, artistic role. If metaphors are ornament, it is far from being a crime, because these are ornaments which, in combining imagination and truth, expand our conceptions of the world by making things *fit* in new ways.

4.2.2 Literature and cognitive structures

Building on the focus on conceptual structures hinted at by metaphors, the attention of more recent work has shifted to the relationship between literature (as part of aesthetic work and eliciting aesthetic experiences) and cognition. This move starts from the limitation of explaining "art for art's sake", and inscribing it into the real, lived experiences of everyday life mentioned above, perhaps best illustrated by the question posed in Jean-Marie Schaeffer's eponymous work—*Why fiction?* (Schaeffer, 1999). Indeed, if literary and aesthetic criticism are to be rooted in the everyday, and in the conventional conceptual metaphors which structure our lives, our brains seem to be the lowest common denominator in our comprehension of both real facts and literary works (Lavocat, 2015).

This echoes our discussion in 3.1.2 of Polanyi's work on tacit knowledge, in which the scientist's knowledge is not wholly and absolutely formal and abstracted, but rather embodied, implicit, experiential. This limitation of

codified, rigorous language when it comes to communicating knowledge, opens up the door for an investigation of how literature and art can help with this communication, while keeping in mind the essential role of the senses and lived experience in knowledge acquisition (i.e. integration of new conceptual structures) (Polanyi & Sen, 2009).

Some of the cognitive benefits of art (pleasure, emotion, or understanding) are not too dis-similar to those posed by Beardsley above, but shift their rationale from strict hermeneutics and criticism to cognitive science. Terence Cave focuses on the latter when he says that literature "*allows us to think things that are difficult to think otherwise*". We now examine such a possibility from two perspectives: in terms of the role of imagination, and in terms of the role of the senses.

Cave posits that literature is an object of knowledge, a creator of knowledge, and that it does so through the interplay between rational thought and imaginative thought, between the "counterfactual imagination" and our daily lives and experiences. Through this tension, this suspension of disbelief is nonetheless accompanied by an epistemic awareness, making fiction reliant on non-fiction, and vice-versa. Working on literary allusions, Ziva Ben-Porat shows that this simultaneous activation of two texts is influenced by several factors. First, the form of the linguistic token itself has a large influence over the understanding of what it alludes to. Its aesthetic manifestation, then, can be said to modulate the conceptual structures which will be acquired by the reader. Second, the context in which the alluding token(s) appears also influences the correct interpretation of such an allusion, and thus the overall understanding of the text. This contextual approach, once again hints at the change of scale that Ricoeur points in his shift from the word to the sentence, and demands that we focus on the whole, rather than single out isolated instances of linguistic beauty. Finally, a third factor is the personal baggage (a personal encyclopedia) brought by the reader. Such a baggage consists of varying experience levels, of quality of the know-how that is to be activated dur-

ing the reading process, and of the cognitive schemas that readers carry with them. Imagination in literary interpretation, builds on these various aspect, from the very concrete form and choice of the words used, to the unspoken knowledge structures held in the reader's mind, themselves depending on varied experience levels. By allowing the reader to project themselves into potential scenarios, imagination allows us to test out possibilities and crystallize the most useful ones to continue building our conception of the fictional world.

The work of imagination also relies on how the written word can elicit the recall of sensations. This takes place through the re-creation, the evocation of sensory phenomena in linguistic terms, such as the *perceptual modeling* of literary works, which can be defined as (linguistic) simulations relying on the senses to communicate situations, concepts, and potential realities, something at work in the process of creating a fiction. This connects back to the modelling complexities evoked in 3.2.2: both source code and literature have at least the overlap of helping to form mental models in the reader.

This attention to the sense calls for an approach of literary criticism as seen through embodied cognition, starting from the postulate that human cognition is grounded in sensorimotoricity, i.e., the ability to feel, perceive, and move. Specifically, pervading cognitive process called perceptual simulation, which is activated when we cognitively process a gesture in a real-life situation, is also recruited when we read about actions, movements, and gestures in texts.

Depicting movement, vision, tactility and other embodied sensations allows us to crystallize and verify the work of the imaginative process. As such, literature unleashes our imaginary by recreating sensual experiences—Lakoff even goes as far as saying that we can only imagine abstract concepts if we can represent them in space¹⁵. It seems that

¹⁵Geoff Hinton, pioneer of modern deep-learning, has reportedly said that, to visualize 100-dimensional spaces, one should first visualize a 3-dimensional, and then "shout 100 really

```
class love{};  
void main(){  
    throw love();  
}
```

Listing 48: Unhandled Love, by Daniel Bezera, published in {code poems} (Bertram, 2012)

the imaginative process depends in part on visual and spatial projections, and suggests a certain fitness of the conceptual structures depicted. By describing situations which, while fictional, nonetheless are possible in a reality often very similar to the one we live in, it is easy for the reader to connect and understand the point being made by the author. So if literature is an object of knowledge, both sensual and conceptual, offering an interplay between rational and imaginative thought, it still relies on the depiction of mostly familiar situations (the protagonists physiologies, the rules of gravity, the fundamental social norms are rarely challenged).

A first issue that we encounter here, in trying to connect source code and computing to this line of thought, is that source code has close to no perceptible sensual existence, beyond its textual form. In trying to communicate concepts, states and processes related to code and computing, and in being unable to depict them by their own material and sensual properties, we once again resort to linguistic processes which enable the bringing-into-thinking of the program text.

The code poem listed in 48 suggests a similar phenomenon when it comes to perceiving motions and sensations through words. The key part of the poem here is the use of the keyword `throw`: as a reserved keyword in some of the most popular programming languages, it is known and has been encountered by multiple programmers, as opposed to a word defined in a specific program (such as a variable name). This previous encoun-

really loud, over and over again", cited in (Akten, 2016)

ters build up a feeling of familiarity and of dread—indeed, the act of the throwing in programming is as dynamic and as violent as in human prose. To throw an object in programming, is to interrupt the smooth execution flow of the program, because something unexpected has happened,—that is, an exception. Additionally, the title of the poem hints at a supplemental implication of the poems motion; any exception that is thrown should be caught, or handled, by another part of the program, in order to gracefully recover from the mishap and proceed as expected. If it's not handled—as is the case in the poem—the program terminates and the source code itself aborts all function.

Vilem Flusser considers poetic thinking as a means to bring concepts into the thinkable, and to crystallize thoughts which are not immediately available to us¹⁶; through various linguistic techniques, poetry allows us to formulate new concepts and ideas, and to shift perspectives. Rendered meaningful via this code poem, a certain conception of love is therefore depicted here as an exception that must be handled (with care) , and the use of a particularly dynamic keyword elicits such a feeling in a reader who previously had to throw and handle exceptions.

Another example of how source code can communicate concepts can be seen in 49. In this case, we can see in the relation between the name of the function, `find` and the three local variables `high`, `low` and `probe`, that the act of finding is going to imply some sort of *search space*. The search space is going to be traversed in an alternating way, called the *binary search* in computer science terms¹⁷.

¹⁶"In this sense we may say that the intellect expands intuitively. We may, however, define the intuition that results in the production of proper names better, since it is a productive intuition. We may call it "poetic intuition." The proper names are taken, through this intuitive activity, from the chaos of becoming in order to be put here (*hergestellt*), that is, in order to be brought into the intellect." (FLUSSER & Novaes, 2014)

¹⁷The author of 49 said of the difference between concept and implementation: "Nothing could be simpler, conceptually, than binary search. You divide your search space in two and see whether you should be looking in the top or bottom half; then you repeat the exercise until done. Instructively, there are a great many ways to code this algorithm incorrectly, and several widely

```

package binary;

public class Finder {
    public static int find(String[] keys, String target) {
        int high = keys.length;
        int low = -1;
        while (high - low > 1) {
            int probe = (low + high) >>> 1;
            if (keys[probe].compareTo(target) > 0)
                high = probe;
            else
                low = probe;
        }
        if (low == -1 || keys[low].compareTo(target) != 0)
            return -1;
        else
            return low;
    }
}

```

Listing 49: Binary search, implemented by Tim Bray in *Beautiful Code* highlights variable names as an indicator of the spatial component of the function's performance (Bray, 2007).

Here, we thus have two indicators, syntactical and structural. First, `high` and `low`, imply the space in-between, a space to be explored via `probe`¹⁸. Second, the use of only two statements inside the `while` loop represents the simplicity of the search process itself, a search process which, as `(high - low > 1)` tells us, implies a shrinking search space¹⁹.

published versions contain bugs." (Bray, 2007)

¹⁸Conversely these variables could have been named `start`, `end` and `current`, with similar purpose, but a different denotation

¹⁹Rather than expliciting checking if the target has been found inside the loop, the code's simplicity relies on the fact that another definition for finding is that of reducing search space: "*Some look at my binary-search algorithm and ask why the loop always runs to the end without checking whether it's found the target. In fact, this is the correct behavior; the math is beyond the scope of this chapter, but with a little work, you should be able to get an intuitive feeling for it—and this is the kind of intuition I've observed in some of the great programmers I've worked with. [...] You could do the math to figure out when the probability of hitting the target approaches 50 percent, but qualitatively, ask yourself: does it make sense to add extra complexity to each step of an O(log₂ N) algorithm when the chances are it will save only a small number of steps at the*

By paying attention to the spatial and embodied implicit meanings held in the syntactic structures used in both literature and source code, we can start to see how a certain sense of understanding being extracted from reading either kind of texts depends on embodiment. In the case of program texts, the point is to reduce computational space into humanly embodied space; similarly, literature engages in communicating different kinds of space.

4.2.3 Words in space

Beyond the use of metaphor, literature allows the reader to engage cognitively with the world of the work, and the interrelated web of concepts that can then be grasped once they are put into words. This process of putting down intention, through language and into written words, is also the process of transforming a time-based continuum (speech) into a space-based discrete sequence; a process called grammaticalization, explored further in (Bouchardon, 2014). This is valid both for human prose and machine languages: the unfathomably fast execution of sequential instructions is manifested as static space in source code.

Literary theory also engages with the concept of space. We have seen in the subsection above that there is a particular attention being given to movement in space, through embodied cognition; in that case, the use of a specific syntax can elicit a kinetic reaction in the incarnated reader. We now pay attention to how spatiality interplays with meaning in literature, looking at the spatial form of the text in general, and to spatio(-temporal) markers in the text in specific.

First, we leave behind some traditional concepts in literary theory. We have seen that, due to source code's non-linearity and collaborative aspect,

end? The take-away lesson is that binary search, done properly, is a two-step process. First, write an efficient loop that positions your low and high bounds properly, then add a simple check to see whether you hit or missed." (Bray, 2007)

concepts such as narrative and authorship are somewhat complicated to map across fields.

We have mentioned above that the fictionality of a text provides a kind of text-based simulation for a combination of events, characters and situations. While source code, by its actual execution, might tend to be classified rather as non-fiction, we nonetheless show here that, by evoking interconnected entities, it also participates to the construction of mental models.

Here, we pay particular attention to fictional space: the web of relationships, connotations and suggestions that hint at a broader world than the one immediately at hand in a work of literature. This fictional space, or *storyworld* is not to be equated to what we have denoted as the problem domain. Rather, it is what exists through, yet beyond, the text itself; we refer to it as the *world of reference*.

To focus on the specific tokens denoting space, we rely on the distinction operated by Marie-Laure Ryan on the topic (Ryan, 2009). The starting point she offers is to consider how the spatial extension of the text, its existence in a certain number of dimensions²⁰ impacts the readers' perception of the narrative.

At the simplest level, we see this illustrated in 50. In this listing, we can see how the most direct spatial perceptions of the program text, its indentation, actually represents semantic properties: the indent on `class_space` is related to it existing at a different level (scope) than the variables `dimensions` and `alone`, just like the indent before `def __init__` differs from the one before `def new_space` also signify changes in lexical scope.

Moving beyond this immediately visual spatial component, Ryan shifts to the spatial form of the text. Rather than looking at the space in which it is deployed, it is considering

a type of narrative organization characteristic of modernism that

²⁰An oral narrative exists in zero dimensions, a live TV news ticker exists in one dimension, a printed or digital page exists in two dimensions, while a theater play exists in three dimensions.

```

class Space:
    class_space = "ever present"

    def __init__(self, dimensions):
        self.dimensions = dimensions
        alone = True

def new_space():
    new = new Space(4)

```

Listing 50: This snippet shows how the spatial extension of the text corresponds to the structural semantics of the code, in the Python programming language.

deemphasizes temporality and causality through compositional devices such as fragmentation, montage of disparate elements, and juxtaposition of parallel plot lines. (Ryan, 2009).

Narrative, in its traditional sense of coherent, sequential events whose developments involve plot and characters, is seldom mentioned in writing source code. In source code, narrative is already deemphasized and the spatial form of the text mentioned above is therefore better suited to match the material of the code. Indeed, Ryan continues:

The notion of spatial form can be extended to any kind of design formed by networks of semantic, phonetic or more broadly thematic relations between non-adjacent textual units. When the notion of space refers to a formal pattern, it is taken in a metaphorical sense, since it is not a system of dimensions that determines physical position, but a network of analogical or oppositional relations perceived by the mind. (Ryan, 2009)

Space, along with interactivity, is a core feature of the digital medium²¹. Janet Murray also puts spatiality as one of the core distinguishing features

²¹As N. Katherine Hayles states in her eponymous essay, "print is flat, code is deep" (Hayles, 2004)

of digital media, at the forefront of which are digital games²².

An example of this intertwining of flat textual screen and spatial depth is the overall genre of interactive fiction, which displays prompts for textual interaction on a screen, accompanied with the description of where the reader is currently standing in the fictional world. Exploration can only be done in a linear fashion, entering one space at a time; and yet the system reveals itself to contain spaces in multiple dimensions, connected by complex pathways and relationships. The listing in 51 shows how the execution processes of a program text can be expressed spatially in the comments, and then textually in the rest of the file. Since comments are ignored by the computer, this depiction is only to help the human reader in their spatial representation of the executed program.

As Murray mentions, these features are not limited to those playful interactive systems presented as works to be explored (be it e-literature or digital games), but are rather a core component of digitality. Beyond the realm of fiction, one can see instances of this in the syntax used in both programming languages and programming environments (see 3.3.2 and our overview of IDEs). For instance, the use of the GOTO statement in BASIC, of the JMP and MOV instructions in x86 Assembly, or the use of the return in the C family of programming languages all hint at movement, at going places and coming back, representing the non-linear perception of program execution²³.

And yet, Ryan hints at another aspect of spatial form specifically in the

²²"The computer's spatial quality is created by the interactive process of navigation. We know that we are in a particular location because when we enter a keyboard or mouse command the (text or graphic) screen display changes appropriately. (Murray, 1998)

²³In the meantime, program execution is still considered to be linear by the machine, since instructions are executed one after the other. The use of multi-core architecture and parallel processing does complicate this picture, but programmers rarely engage directly with the specification of which CPU core executes which instruction. What they do engage with, is parallel programming, in which things happen simultaneously, thus presenting cognitive complexity insofar as two processes being run in parallel imply some sort of distinct semantic spaces to be reflected in the mental model of the programmer.

```

* Part 1 -- Initial checks
*
*      * . called by
*      |   MAC clients
*      v
*      +-----+ +-----+ : +-----+
*      +=====+=====
*      | mac_tx | ->| device    | -*-->| mac_protect_check | ->v Is this
*      |        |    v
*      +-----+ | quiesced? |    +-----+ v case? See
*      [1]      v
*      +-----+
*      +=====+=====
*      |          * . Yes           * failed
*      |          v               | frames
*      +-----+
*      +-----+
*      | freemsgchain | <-----+           Yes . *
*      No . *
*      +-----+
*      v
*      +-----+
*      | goto |
*      | SRS TX |
*      | func |
*      +-----+
*      |
*      v
*      +-----+
*      +-----+ return |
*      | cookie |
*      +-----+

```

Listing 51: This listing includes an execution flow diagram inside the program text itself, testifying to the inherently fragmented and non-linear execution of source code. (Mustacchi, 2019)

digital medium:

But an even more medium-specific type of spatial form resides in the architecture of the underlying code that controls the navigation of the user through a digital text. (Ryan, 2021)

As writers and readers of this architecture, of which source code is the blueprint, we gather information through syntax about developments in space and time into a cognitive map or mental model of narrative space²⁴.

Mental maps are therefore dynamically constructed in the course of reading and consulted by the reader to orient herself in the program. A very simple example of spatialization of meaning, both visually and conceptually, can be seen in 52. There, the spatial component is rendered specifically through the syntax of HTML. HTML, as a markup language, has a specific ontology: it is fundamentally made up of elements who contain other elements, or are self-contained. When an element is contained into another, a specific semantic relationship occurs, where the container influences the contained, and vice-versa. Therefore, what we see at first is layout spatialization, which leads to this specific triangle shape. By using the semantics of the language, in which certain elements can only exist in the context of others, this layout spatialization²⁵ also comes to delimit certain semantic areas. This explicitly poetic example takes religion, and the representation of God as its problem domain; its expressive force comes by describing it as both the all-including and the all-included, and thus escaping the implicit rules of everyday spatiality, that a thing cannot contain itself.

A more concrete example can be seen in 53. Written in the style of soft-

²⁴The term *narrative* is used here to describe the effective behaviour of the program, once executed. Since source code appreciation is subject to its function, following the narrative of source code would then amount to following its correct execution path(s), even though *description* fits better to most program texts since, from the machine perspective, it describes exactly what it is doing.

²⁵While not functionally necessary, the indents added to the listing further highlight the computational concept of nestedness through visual cues.

```
<GOD>
  <universe>
    <galaxy>
      <solarsystem>
        <earth>
          <island>
            <town>
              <garden>
                <flowerbed>
                  <snowdrop>
                    <petal>
                      <molecule>
                        <proton>
                          <quark>
                            <GOD>
                            </quark>
                          </proton>
                        </molecule>
                      </petal>
                    </snowdrop>
                  </flowerbed>
                </garden>
              </town>
            </island>
          </earth>
        </solarsystem>
      </galaxy>
    </universe>
  </GOD>
```

Listing 52: Nested, by Dan Brown and published in {code poems} (Bertram, 2012)

```

func (h *http2Listener) Shutdown(ctx context.Context) error {
    pollIntervalBase := time.Millisecond
    nextPollInterval := func() time.Duration {
        // Add 10% jitter.
        //nolint:gosec
        interval := pollIntervalBase + time.Duration(weakrand.J
            & .Intn(int(pollIntervalBase/10)))
        // Double and clamp for next time.
        pollIntervalBase *= 2
        if pollIntervalBase > shutdownPollIntervalMax {
            pollIntervalBase = shutdownPollIntervalMax
        }
        return interval
    }

    timer := time.NewTimer(nextPollInterval())
    defer timer.Stop()
    for {
        if atomic.LoadUint64(&h.cnt) == 0 {
            return nil
        }
        select {
        case <-ctx.Done():
            return ctx.Err()
        case <-timer.C:
            timer.Reset(nextPollInterval())
        }
    }
}

```

Listing 53: This listing represents the various steps taken in order to shutdown a HTTP server, and shows multiple aspects of spatio-temporal complexities (WeidiDeng, 2023)

ware engineers, rather than poets, this listing describes a function which gracefully shuts down a HTTP server. Essentially, the function `Shutdown()` regularly checks if the number of connections to the server is zero. If it reaches zero, it considers the process completed without errors; it waits until it receives an error from the context, or if it receives a tick from a timer setup in advance.

The first reference we can look at is mostly spatial, and takes place at the declaration of `nextPollInterval`. By being another function declaration, it is both self-contained, but also has access to variables in its declarative environment, such as `pollIntervalBase`. A long, dynamic series of state-

ments which double a timer interval everytime it is called is thus compressed into a single token, `nextPollInterval`, and can then be passed as an argument to timer functions. Here, the space of the timer interval's calculation is compressed and abstracted away.

Interestingly, we can note the comment `// Add 10% jitter`, which explains the calculation of the subsequent interval. The word *jitter* usually refers to a quicky, jumpy movement, but is here used to facilitate the understanding of adding a random number to the previous one, effectively deviating the timer from its linear increase. Here, using the word *jitter* immediately evokes feeling of small, unpredictable change.

The second reference is primarily temporal. The keyword `defer` in the line `defer timer.Stop()` specifically marks the deferred execution of this particular function to the specific moment at which the current function (`Shutdown()`) returns. This reference is not absolute (as is the timer on the line above, even though it might not be determinate), but rather relative, itself dependent on when the current function will return. Here, the programming language itself makes it simple to express this relative temporal operation.

Finally, we can take a look at both the last `select` statement of the function to see a more complex interplay of both space and time. There are two things happening there. With the specific `<-` arrow, the pictorial representation shows how a message is incoming, either from `ctx.Done()`, which itself comes from outside the current function, given as an argument, or from `timer.C`, which comes from the timer that has just been declared in the current function. Both of these messages come from different places, one very distant, and the other very local, and might arrive at different moments. Here, the `<-` denotes the movement of an incoming message, expliciting where the messages come from, and in which order they should be treated, and thus facilitates the handling of event with varying spatio-temporal properties.

The listing 53 shows not only different spaces of executions, nor only

different moments of execution, but very much the intertwining of space and time. One of the earlier approaches to the specific tokens which represent space in the traditional novel has also related it to time: the chrono-topes is described by Mikhail Bakhtin as the tight entanglement of temporal and spatial relationships that are artistically expressed in literature. Those markers execute a double function, as they allow for the reification of temporal events and spatial settings during the unfolding of narrative events²⁶.

While Bakhtin introduces the concept from a marxist-historical point of view, analyzing notions of history, ideal, epics and folklore through that lense, it is nonetheless useful for our purposes. Chronotopes are a kind of marker which enable the understanding of where something comes from (such as an explicit module declarations in header files, or inline before a function call), or when something should happen (such as the `async/await` keyword pair in ECMAScript denoting the synchronicity of an operation or the `defer` keyword indicating that a specified function will only be called *when* the current function returns).

Thus, the chronotopes give flesh to the events described in (and then executed from) source code. As such, they function as the primary means of materializing time in space. From a network of these chronotopes, along with metaphors and other devices that are explicated in 5.2, emerges a concretization of representation which the reader can use to constitute a mental model of the program text.

Syntactical literary devices allow readers to engage cognitively with a particular content; they enable the construction of mental models a particular narrative, through a network of metaphors, allusions, ambiguous

²⁶"Time, as it were, thickens, takes on flesh, becomes artistically visible; likewise, space becomes charged and responsive to the movements of time, plot and history." (Bakhtin, 1981)

interpretations and markers of space and time. We have shown that these literary devices also apply to source code, especially how the use of machine tokens and human interpretation suggest an aesthetic experience through metaphors, and with particular markers that are needed to make sense of the time and space of a computer program, which differs radically from that of a printed text. This making sense of a foreign time and space is indeed essential in creating a mental map of the storyworld (in fiction) or the world of reference (in non-fiction).

The use of the term map also implies a specific kind of territory, enabled by the digital. As a hybrid between the print's flatness and code's depth, Ryan and Murray—among many others—identify the digital narrative as a highly spatialized one. This feature, Ryan argues, is but a reflection of the inner architecture of source code. Pushing this line of thought further, we now turn to architecture as a discipline to investigate how the built environment elicits understanding, and how such possibilities might translate in the space of program texts.

4.3 Architecture and understanding

At its most common denominator, architecture is 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, and includes a subjective perspective (Lynch, 1959). Moving from the urban planner's perspective to the architects, we see how each individual component contributes to the overall legibility of the system. This section

considers how individual structures, through their assessed beauty, offer a cognitive involvement to their participants.

Beauty in architecture is one of the discipline's fundamental components, dating back to Vitruvius's maxim that a building should exhibit *firmitas, utilitas, venustas*—solidity, usefulness, beauty. And yet in practice, beauty, or the ability to elicit an aesthetic experience, is not sufficient, and sometimes not even required, for a building to be considered architectural. Even though architecture is usually considered as an art, it is also a product of engineering, and thus a hybrid field, one where function and publicness modulate what could be otherwise a "pure" aesthetic judgment.

This section looks at architecture through its multiple aspects, to highlight to which extent some of these are reflected in source code²⁷. Through an investigation of the tensions and overlaps of form, function, context and materiality in the built space, we identify similarities in the programmed space. Particularly, we will look at how an understanding of patterns translates across both domains, in response to both architecture and programming's material constraints, due to the physical instantiation of buildings and programs in a situated context.

4.3.1 Form and Function

Particularly, our interest here is with the cognitive involvement in the architectural work. What is there to be understood in a building, and how do buildings make it intelligible? The early theoretical answers to this question is to be found in the work of Italian architects, such as Andrea Palladio, whose conception of its discipline came from ideal platonic form, and mathematical relation between facade and inner elements, as well as Leon Battista Alberti, whose consideration of beauty in architecture, as such an organization of parts that nothing can be changed without detriment to

²⁷Recall how, in 2.3.3, programmers tended to refer extensively to themselves as architects, engineers or craftspeople.

the whole (Scruton, 2013)²⁸.

While structure is meant to stand the test of time and natural forces²⁹, utility can be assessed by the extent to which a building fulfills its intended function. How the beauty of a building relates to its function, whether it can be completely dissociated from it, or if it is dependent on the fulfillment of its function, is still a matter of debate between formalists and functionalists. Nonetheless, the position we take here is in line with Parsons and Carlson, in that fitness of an object is a core component of how it is appreciated aesthetically (Parsons et al., 2012), and that form is hardly separable from function.

In some way, then, form should be able to communicate the function of a building. Roger Scruton, in his philosophical investigation of architecture, brings up the question of language—if buildings are to be cognitively engaged with, then one should be able to grasp what they communicate, what they stand for, what they express. To do so, he starts from the fact that architectural works are often composed of interconnected, coherent sub-parts, which then contribute to the whole, in a form of *gestaltung*.

Architecture seems, in fact, to display a kind of 'syntax': the parts of a building seem to be fitted together in such a way that the meaningfulness of the whole will reflect and depend upon the manner of combination of its parts. (Scruton, 2013)

Yet, he develops an argumentation which suggests that architecture is not so much articulated as a language, than as a set of conventions and rules, and that it is not a representative medium (which would imply valid and invalid syntax, as well as intent), but rather an expressive one. Architectural significance, then, relies on the presence and arrangement of

²⁸Such a definition is a reminiscent of how Vladimir Nabokov defines beauty in literature: "A really good sentence in prose should be like a good line in poetry, something you cannot change, and just as rhythmic and sonorous" (Nabokov, 1980)

²⁹A purpose exemplified by the still standing structures of Roman and Greek antiquities, resulting from a particular mixing process of concrete.

those evolving conventions—that is, a style—rather than on the depiction of a subject through an exact syntax. While architecture might not represent content the same way literature does, it is nonetheless expressive, and relies on particular styles—recurring formal patterns and ways of doing—to express a tone, a feeling, or a *stimmung* in their dwellers.

As identified in 2.3.3, the similarities between software and architecture can be mapped as symmetrical approaches: as top-down or bottom-up, from an architect's perspective, or from a craftperson's. Since we focus on what a building expresses, we need to consider the source of such an expression. First, we look at how modernism, and the conventions that make up this architectural thought, are the top-down result of the intersection of function, form and industry, and reveal the influence of functional design on the aesthetic appreciation of a work.

The central modern architectural standard is Louis Sullivan's maxim that *form follows function*, devised as he was constructing the early office buildings in North America. Sullivan's statement is thus that what the building enables its inhabitants to do, inevitably translates into concrete, visible, and sensual consequences.

All things in nature have a shape, that is to say, a form, an outward semblance, that tells us what they are, that distinguishes them from ourselves and from each other ...It is the pervading law of all things organic and inorganic, of all things physical and metaphysical, of all things human and all things superhuman, of all true manifestations of the head, of the heart, of the soul, that the life is recognizable in its expression, that form ever follows function. This is the law. (Sullivan, 1896)

The value of the building is therefore derived from what it allows the individuals to do: the office building allows them to work, the school to learn, the church to pray and the house to live. To do so, modernist architecture rejects any superfluous decoration, or extraneous addition, as

a corruption of the purity of the building's function. In a similar vein, Le Corbusier, another fundamental actor of modern architecture, equates the building with its function, advocating for the suppression of decorative clutter and unnecessary furnishings and possessions, and hailing transparency and simplicity as architectural virtues (Le Corbusier & Saugnier, 1923), and culminating in Le Corbusier's assessment that the architectural plan as a generator, and the house as a machine to be lived in.

From this perspective, architectural works are a kind of system, in that they constitute sets of interrelated structural components, where the parts are connected by distinctive structural and behavioral relations; and yet the set of conventions to which Le Corbusier contributes is an abstract representation of this systemic nature. He focuses on the plan as the primary source of architectural quality. For software developers, the equivalent of an architectural plan would be a modelling system such as UML: a language to describe structural relationships between software components, with an example shown in 4.1. From a modernist angle, the aesthetic value of a building is thus directly dependent on how well it performs an abstractly defined function for its users, assessed at a structural level.

Just as a two-dimensional floorplan and a three-dimensional building are different, a diagram and a program text are also different. This difference is highlighted through the process of construction in architecture, and implementation in software development, involving respectively engineers and programmers to realize the work that has been designed by the architect.

It is clear the modernists thought of function as engineering function, and aligned it with engineering aesthetics³⁰. Nonetheless, such a conception of function is definitely machinic, consisting of airflow, heat exchange or drainage, expressing a particular feeling of progress and achievement through industrial manufacturing techniques allowing for new material

³⁰*Esthétique de l'ingénieur* is the title of one of the chapters of Le Corbusier's manifesto, *Vers une Architecture* (Le Corbusier & Saugnier, 1923)

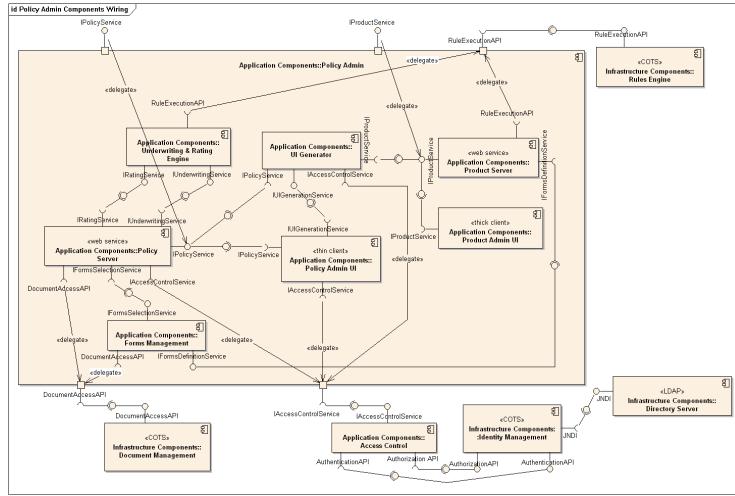


Figure 4.1: Description of a software component and its inner relations in the Universal Modelling Language, (Wikipedia, 2023b)

capabilities against contextual understandings. Here again, the human is but a small part in a dynamic system.

Jacques Rancière, in his study of the Werkbund and the Bauhaus-inspired architecture, offers an alternative approach, away from the strict functionality laid out by Sullivan and Le Corbusier before him. The simplification of forms and processes, he writes of the AEG Turbinenhalle in Berlin, which is normally associated with the reign of the machine, finds itself, on the contrary, related to art, the only thing able to spiritualize industrial work and common life (Ranciere, 2013).

By paying attention to the role of a detail, and of the human subjectivity and situatedness of the people inhabiting the building, departs from the strict function of an object or of a building, to its actual use. Such a shift moves the aesthetic judgment from a structure-centric perspective (such as Le Corbusier's ideal dimensions), to a human-centric perspective (such as Lacaton & Vassal's practical extension of space and light). Peter Downton reiterates this point, when he states that "*buildings and design are often judged from artistic perspectives that bear no relation to how the building's*

occupants perceive or occupy the building." (Downton, 1998); his conception of the artistic here, is one that aligns with Kant's definition of a work that is purposive in itself, and not based on a function that it should fulfill.

One can see a translation of such a self-referential conception of art in the class of building which encompass follies and pavillions. These kinds of buildings are constructed first and foremost for their decorative properties, and only secondarily for its structural and functional properties. Follies, for instance, are individual buildings built on the demand of one specific individual's desire. They aim to represent something else than what they are, with no other purpose than ornament and the display of wealth. Pavillions, in the modern acceptation of the term, are rather displays of architectural and engineer prowess, demonstrating the use of new techniques and materials. By focusing only on design and technical feat, it is this prowess itself that is being represented: the function of the building is only to represent the skill of its builders. For instance, Junya Ishigami's pavillion at the Venice Biennale in 2008, shown in 4.2 consisted in a very elegant and aerial structure, but whose function was depending on the fact that no living being interacted with it³¹.

As an artform, architecture provides an immersive and systemic physical environment, and thus shapes human psychology and agency within it, in turn forcing the dweller to acknowledge and engage with their environment. This suggests that, from a formal, top-down approach which considers architecture as possessing a systematic language to be realized exactly at a structural level, there exists a complementary, bottom-up approach, centered around human construction and function.

³¹Indeed, the structure collapsed due to a cat's playfulness: "The Barbican says that the 37-year-old Ishigami is "internationally acclaimed", and there is certainly a buzz and fascination around him. Last year he won the Golden Lion, the highest prize at the Venice Architecture Biennale, for a structure that collapsed almost as soon as it was built, following an accident with a cat. Little was left but a scrawled note saying "Scusate, si è rotto. I'm sorry It's broken." (Moore, 2011)"



Figure 4.2: Pavillion built by Junya Ishigami + associates, showing a focus on appearance and structural feat, rather than habitability. Picture courtesy of Iwan Baan, 2008.

4.3.2 Patterns and structures

A counterpoint to this modernist approach of master planning is that of Christopher Alexander. Along with other city planners in the United States, such as William H. Whyte or Jane Jacobs, Alexander belongs to an empirical tradition of determining what makes a built environment *good* or not, by examining its uses and the feelings it elicits in the people who tread its grounds. He elaborates an approach to architecture which does not exclusively rely on abstract design and technological efficiency, but rather takes into account the multiple layers and factors that go into making

[...] beautiful places, places where you feel yourself, places where you feel alive (Alexander, 1979) [...]

In *The Timeless Way of Building*, he focuses on how beauty is involved in moving from disorganized to organized complexity, a design process which is not, in itself, the essence of beauty, but rather the condition

for such beauty to arise. Alexander's conception of beauty, while very present throughout his work, is however not immediately concerned with the specifics of aesthetics, but rather with the existence of such objects. This existence, in turn, does require to be experienced sensually, including visually.

In this process of achieving organized complexity, he highlights the paradoxical interplay between symmetry and asymmetry, and pinpoints beauty as the "deep interlock and ambiguity" of the two, a beauty he also finds in the relationship between static structures of the built environment, and the flow of living individuals in their midst. In his perspective, then, architecture should take into account the role of tension between opposite elements, rather than the combination of rational and abstract design elements. Such an approach echoes other considerations of tension as a source for stimulating human engagement, such as Ricoeur's analysis of the metaphor (see 4.2.1), and the resolution of the riddles presented in works of obfuscated source code (see 2.1.2).

He therefore considers a possible aesthetic experience as a consequence of qualities such as appropriateness, rightness to fit, not-simplistic and wholeness. All of these have in common the subsequent need for a purpose, a purpose which he calls the *Quality Without a Name* (Alexander, 1979). This quality, he says, is semantically elusive, but nonetheless exists; it is, ultimately, the quality which sustains life, a conclusion which he reached after extensive empirical research: no one can name it precisely, but everyone knows what it refers to. It is the quality which makes one feel at home, which makes one feel like things make sense in a deep, unexplainable way³². This reluctance to being linguistically explicated is echoed in

³²"It is always looking at two entities of some kind and comparing them as to which one has more life. It appears to be a rank bit of subjectivity. [...] It turns out that these kind of measurements do correlate with real structural features in the thing and with the presence of life in the thing measured by other methods, so that it isn't just some sort of subjected I groove to this, and I don't groove to that and so on. But it is a way of measuring a real deep condition in the particular things that are being compared or looked at." (Alexander, 1996)

the work of the craftsman, where a practitioner often finds herself showing rather than telling (Pye, 2008), another domain with which software developers identify, explicated in 2.3.3.

Among the adjectives used to circle around this quality are whole, comfortable, free, exact, egoless, eternal (Alexander, 1979). Some of these qualities can also be found in software development, particularly wholeness and comfort. A whole program is a program which is not missing any features, whose encounter (or lack thereof) might cause a crash. If if a function implies a systematic design, such systematic design is not compromised by the lacking of some parts. Conversely, it is also a program which does not have extraneous—useless—features.

A comfortable program text being is a program which might be modified without fear of some unintended side-effects, without invisible dependencies which might then compromise the whole. There is enough separation of concerns to ensure a somewhat safe working area, in which one can engage in epistemic probing assuming that things will not be breaking in unexpected ways; being whole, it also provides a higher sense of meaning by realizing how one's work relates to the rest of the construction. The implication here is that comfort derives from a certain kind of knowledge, a knowledge of how things (spatial arrangements, technical specifications, human functions) are arranged, how they relate to each other, how they can be used and modified.

To complement this theoretical pendant, Alexander conducted empirical research to find examples of such qualities, in a study led at the University of Berkley which resulted in his most popular book, *A Pattern Language* (Alexander et al., 1977). In it, he and his team lists 253 patterns which are presented as to form a kind of language, akin to a Chomskian generative grammar, re-usable and extendable in a very concrete way, but without a normative, quasi-biological component. It turns it out that such a documentation, of re-usable configuration and solutions for contextual problem-solving, had a significant echo with computer scientists.

A whole field of research developed around the idea expressed in *A Pattern Language*, at the crossover between computer science and architecture³³ 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 enough to be shared with others. Patterns enable a cognitive engagement based on a feeling of familiarity, and of recognizing affordances.

Extending from the similarities of structure and function 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 language³⁴, then with C++, until today, as most of the programming languages in 2023 include some sort of object-orientation and encapsulation. What object-orientation does, is that it provides a semantic structure to the program, reflected in the syntactic structure: objects are conceptual entities, with states and actions, as discussed in 3.2.2 and shown in 45. This enables such objects to be re-used within a program text, and even across program texts.

The similarities between a pattern and an object, insofar as they are self-contained solutions to contextual situations that emerged through practice, and resulting from empirical deductions, caught on with software developers as a technical solution with a social inflection, rather than a computational focus. Writing in *Patterns of Software*, with a foreword by

³³See, for instance, the *Beautiful Software Initiative* as an organized effort to develop Alexander's theses on growth, order, artefact and computation (Bryant, 2022).

³⁴For an extensive history of the design and development of the Smalltalk hardware and software, see Alan Kay's *Early History of Smalltalk* (Kay, 1993).

Alexander, Richard P. Gabriel addresses this shift from the machine to the human:

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)

The real issue raised here in programming seems to be, again, not to speak to the machine, but to speak to other humans. The programming paradigm of object-orientation aims at solving such complexity in communication. While understanding software is hard, creating, identifying, and formalizing patterns into re-usable solutions turns out to be at least as hard (Taylor, 2001). Part of this comes from a lack of visibility of code bases (most of them being closed source), but also from the series of various economic and time-sensitive constraints to which developers are subject to (and echoes those in the field of architecture), and which result in moving from making something great to making something good enough to ship. The promise of software patterns seemed to offer a way out by—laboriously—codifying know-how. Interestingly, while the increase in software quality has been found to result from the application of engineering practices (Hoare, 1996), the discovery and formalization of the software patterns takes place through the format of writers' workshops³⁵,

³⁵As taken from the website of the 2022 Pattern Languages of Programming conference: "At PLoP, we focus on improving the written expression of patterns through writers' workshops. You will have opportunities to refine and extend your patterns with the assistance of knowledgeable

presenting a different mode of knowledge transmission.

Throughout his work, Gabriel draws from the work of an architect to weave 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 a pattern languages of built concrete or abstract structures. We develop further two concepts in particular, and show how *habitability* and *compression* enable an understanding of such structures.

Compression and habitability in functional structures

We have seen how source code is an inherently spatial medium, with entrypoints, extracted packages, parallel threads of executions, relative folders and directories and endless jump between files. Reading a program text therefore matches more closely an excursion into a foreign territory whose map might be misleading, than reading a book from start to finish. For instance, 4.3 builds on a longer history of using the city as a metaphor for large code bases, and visualizes classes, packages and version in three dimensions.

Given this somewhat literal mapping of source code structure onto urban structure, and given the abstract structure of object-oriented code, a reader of source code will need to find their bearings and orient themselves³⁶. Once the entrypoint is found, the programmer starts to explore the programmed maze and attempts to make sense of their surroundings, as a step towards the construction of mental models.

Both inhabitants in a building and programmers in a code base have a tendency to be there to *accomplish something*, whether it might be living,

and sympathetic patterns enthusiasts and to work with others to develop pattern languages."
(Guerra & Manns, 2022).

³⁶"Exploring a source code repository always starts with finding out what the OS will select as the entry point. 99% of the time it means finding the 'int main(int,char**) function" says Fabien Sanglard on the topic of reverse-engineering code-bases (Sanglard, 2018).

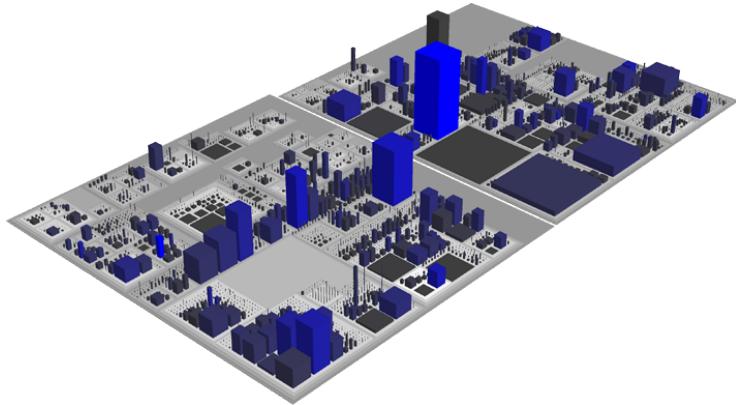


Figure 4.3: CodeCity is an integrated environment for software analysis, in which software systems are visualized as interactive, navigable 3D cities. The classes are represented as buildings in the city, while the packages are depicted as the districts in which the buildings reside. (Wettel, 2008)

working or eating for the former, or fixing, learning or modifying for the latter.s Particularly in software, one of the correlated functions of a program text is to be maintainable; that is, it must be made under the assumption that others will want to modify and extend source code. Other pieces of code might just be satisfying in being read or deciphered (as we've seen in source code poetry in 2.3.1 or with hackers in 2.1.2) but this assumption of interaction with the code brings in another concept, that of *habitability*. In Gabriel's 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. For instance, writing in the ACM Queue, an anonymous programmer discusses the beauty in a code where the separation between which

sections of the source are hardware-dependent and which are not, as seen in 54. In that example, it is clear to the programmer what the problem-domain is: counter incrementation, high-performance computation, or a specific Intel chip.

There are several things which we can identify here. First, the three lines at the top of the listing indicate version numbers, which do not hold any computational functionality, but rather a human functionality: it communicates that this software considers change and evolution as core part of its source code, inviting the programmer reader to further modify it³⁷

Second, the line defining the types of CPUs supported by the software is written in human-intelligible way, rather than a cryptic hexadecimal notation³⁸. While the CPUs are ultimately represented in hexadecimal notation, the effort is made to render things intelligible to and quickly retrievable from the programmer's memory.

Finally, the `struct pmc_mdep` is a shorthand notation for "machine-dependent". In an era in which software can theoretically be executed on different hardware architectures, it is welcome to make the difference between the variables themselves, which apply across platform, and the values of these variables, which need to be changed per platform³⁹. This is

³⁷From the anonymous programmer: "*The engineer clearly knew his software would be modified not only by himself but also by others, and he has specifically allowed for that by having major, minor, and patch version numbers. Simple? Yes. Found often? No.*" (Vicious, 2008).

³⁸"*Nothing is more frustrating when working on a piece of software than having to remember yet another stupid, usually hex, constant. I am not impressed by programmers who can remember they numbered things from 0x100 and that 0x105 happens to be significant. Who cares? I don't. What I want is code that uses descriptive names. Also note the constants in the code aren't very long, but are just long enough to make it easy to know in the code which chip we're talking about.*" (Vicious, 2008).

³⁹"*It would seem obvious that you want to separate the bits of data that are specific to a certain type of CPU or device from data that is independent, but what seems obvious is rarely done in practice. The fact that the engineer thought about which bits should go where indicates a high level of quality in the code.*" (Vicious, 2008).

```

#define PMC_VERSION_MAJOR      0x03
#define PMC_VERSION_MINOR      0x00
#define PMC_VERSION_PATCH      0x0000

/* * Kinds of CPUs known */

#define __PMC_CPUS() \ __PMC_CPU(AMD_K7, "AMD K7") \
→  __PMC_CPU(AMD_K8, "AMD K8") \ __PMC_CPU(INTEL_P5, "Intel \
→  Pentium") \ __PMC_CPU(INTEL_P6, "Intel Pentium Pro") \
→  __PMC_CPU(INTEL_CL, "Intel Celeron") \ __PMC_CPU(INTEL_PII, \
→  "Intel Pentium II") \ __PMC_CPU(INTEL_PIII, "Intel Pentium III") \
→  \ __PMC_CPU(INTEL_PM, "Intel Pentium M") \ __PMC_CPU(INTEL_PIV, \
→  "Intel Pentium IV")

// ...

/*
 * struct pmc_mdep
 *
 * Machine dependent bits needed per CPU type.
 */
struct pmc_mdep
{
    uint32_t pmd_cputype; /* from enum pmc_cputype */
    uint32_t pmd_npmc;   /* max PMXs per CPU */
    uint32_t pmd_npmc;   /* PMC classes supported */
    struct pmc_classinfo pmd_classes[PMC_CLASS_MAX];
    int pmd_nclasspmcs[PMC_CLASS_MAX];

    /*
     * Methods
     */
    int (*pmd_init)(int _cpu); /* machine dependent initialization*/
    int (*pmd_cleanup)(int _cpu) /* machine dependent cleanup */
}

```

Listing 54: This header file defines the structure of a program, both in its human use, in its interaction with hardware components, and its decoupling of hardware and software elements.

a good example of a separation of concerns: it is made clear which parts of the program text the programmer needs to pay attention to, and can change, and which parts of the program texts she needs not be concerned with. For a further example of separation of concerns, one could point a beautiful commit is a commit which adds a significant feature, and yet only change the lines of the code that are within well-defined boundaries (e.g. a single function), leaving the rest of the codebase untouched, and yet affecting it in a fundamental way.

Habitability, then, is a combination of acknowledgment by the writer(s) to the reader(s) of the source, by referring to the evolution over time of the software, along with the use of intelligible names and separation of concerns. This distinction relates to Alexander's property of comfort, by affording involvement instead of estrangement. Still, such a feature of habitability, of supporting life, doesn't specify at all what it could, or should, look like. Rather, we get from Alexander a negative definition:

The details of a building cannot be made alive when they are made from modular parts...And for the same reason, the details of a building cannot be made alive when they are drawn at a drawing board. (Alexander, 1979)

If modularity itself is at odds with making good (software) constructions, then its implementation under the terms of an object-oriented programming paradigm becomes complicated. Indeed, the technical formalization of the field came with the release of the *Design Patterns: Elements of Reusable Object-Oriented Software* book, which lists 23 design patterns implementable in software (Gamma et al., 1994). Its influence, in terms of copies sold, and in terms of papers, conferences and working groups created in its wake, is undeniable, with Alexander himself giving a keynote address at the ACM two years after the release. It has, however, been met with some criticism.

Some of this criticism is that patterns are "external", they look like

they come from somewhere else, and are not adapted to the code. In this sense, this corroborates Alexander in being wary of constructions which do not integrate fully within their environments, which do not, in an organic sense, allow for a piecemeal growth⁴⁰. If patterns express relations between contexts, problems and solutions, then it seems that one of the main complaints of developers is that they might, one day, look at the code they were working on and see chunks of foreign snippets dumped in the middle to fix some generic problem, with no understanding for specifics, nor fit in the existing structure. This is judged negatively due to its lack of understanding of context offered by those proposed solutions. In this, blindly applying patterns from a textbook might be a solution, but it's not an elegant one. This criticism also finds its echoes in the Scruton's analysis of architectural styles; rules and conventions, while present in architecture, are often adopted only to be departed from—re-interpreted and adapted to the context of the building (Scruton, 2013).

One aspect that has been eluded so far is therefore that of the programming languages used by the programmer. Indeed, one doesn't write Ruby like one writes Java, C++, or Lisp. If materiality is a core component of eliciting an aesthetic experience in an architectural context, then programming languages are the material of source code, and offer a specific context to the writing and reading of the program text.

A final criticism to software patterns is that they are language-independent. As such, they are often workarounds for features that a particular programming language doesn't allow from the get-go, or offers simpler implementations than the pattern's⁴¹.

While patterns might operate at a more structural level, hinting at different parts of code, and its overall organization, one can also turn to a

⁴⁰Addressing this concern, the failure of strict top-down hierarchies in software development resulted in the *agile* methodology for business teams, now one of the most popular ways of building software products.

⁴¹For instance, Peter Norvig highlights that most patterns in the original book have much simpler implementations in Lisp than in C++ or Smalltalk (Norvig, 1998)

more micro-level. What can a detail do in our understanding of structures? Sometimes decried, sometimes praised in architecture, the detail fulfills multiple roles: acting as a meaningful interface, compressing meaning and testifying for materiality.

Both Scruton and Rancière mention the detail as an essential architectural element. Without contributing to the structural soundness of the construction, it nonetheless contributes to its expressiveness. A blend of the cognitive and sensual is also characteristic of Scruton's "imaginative perception", understood as the perception of the details of built structures, and their extrapolation into the imaginary. Indeed, the experience of the user is based on the points at which it sensually grasps its environment: the detail is therefore the point of interaction between the human and the structure. This imagination depends on the interpretative choices in parsing ambiguous or multiform aspects of the built environment. The detail contributes to the stylistic convention of the creation:

Convention, by limiting choice, makes it possible to 'read' the meaning in the choices that are made ...for style is used to 'root' the meanings which are suggested to the aesthetic understanding, to attach them to the appearance from which they are derived. (Scruton, 2013)

With many external constraints, due to both context and function, the architect or builder does not have much room for personal expression, and it is through details that their intent and their style are being shown. The significance of a detail can be in explaining which conventions the structure adopts, as well as communicating the intent of the creator. A significant detail manages to compress meaning into a restricted physical surface.

Compression is a concept introduced by Gabriel in response to pattern design. In narrative and poetic text, it is the process through which a word is given additional meaning by the rest of the sentence. In a sentence such

as "*Last night I dreamt I went to Manderley again.*" (Du Maurier, 1938), 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, 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 of 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 idiosyncracy, is a thin line of being self-explanatory without being verbose.

For instance, Casey Muratori explores the process of compression in refactoring a program text, first by distinguishing semantic compression from syntactic compression⁴², and then honing in on what makes a compression successful⁴³. Transitioning from uncompressed code, shown in 55 to compressed code, shown in 56, allows the programmer to understand

⁴²"Like, literally, pretend you were a really great version of PKZip, running continuously on your code, looking for ways to make it (semantically) smaller. And just to be clear, I mean semantically smaller, as in less duplicated or similar code, not physically smaller, as in less text, although the two often go hand-in-hand." (Muratori, 2014)

⁴³"Ah! It's like a breath of fresh air compared to the original, isn't it? Look at how nice that looks! It's getting close to the minimum amount of information necessary to actually define the unique UI of the movement panel, which is how we know we're doing a good job of compressing. (Muratori, 2014)

```

int num_categories = 4;
int category_height = ypad + 1.2 * body_font->character_height;
float x0 = x;
float y0 = y;
float title_height = draw_title(x0, y0, title);
float height = title_height + num_categories * category_height + ypad;
my_height = height;
y0 -= title_height;

{
    y0 -= category_height;
    char *string = "Auto Snap";
    bool pressed = draw_big_text_button(x0, y0, my_width,
                                       category_height, string);
    if (pressed)
        do_auto_snap(this);
}

{
    y0 -= category_height;
    char *string = "Reset Orientation";
    bool pressed = draw_big_text_button(x0, y0, my_width,
                                       category_height, string);
    if (pressed)
    {
        // ...
    }
}
// ...

```

Listing 55: genalloc.c, Basic general purpose allocator for managing special purpose memory from the Linux Kernel, displaying examples of source-code spatiality (Muratori, 2014)

broad patterns about the overall architecture of the program text—here, the function is to display a clickable panel on a user interface.

The difference we can see between the compressed and uncompressed goes beyond the number of lines used for the same functionality. A first clue in terms of semantics is the use of strictly syntactic block markers: { and }. There are here strictly to delimitate a code block, with no semantic meaning to the computer. While the uncompressed listing shows all the separate elements needed for a button to exist (such as `x0`, `y0`, `my_height`, etc.), while the compressed listings as encapsulated them into an object

```

Panel_Layout layout(this, x, y, my_width);
layout.window_title(title);

layout.row();
if(layout.push_button("Auto Snap")) {
    do_auto_snap(this);
}

layout.row();
if(layout.push_button("Reset Orientation"))
{
    // ...
}

// ...
layout.complete(this);

```

Listing 56: genalloc.c, Basic general purpose allocator for managing special purpose memory from the Linux Kernel, displaying examples of source-code spatiality (Muratori, 2014)

called `Panel_Layout`, thus abstracting away from the programmer's mind the details of such a panel. This encapsulation then enables a further compression of the program: by adding the `push_button()` method on the `layout`, the compressed code realizes the same functionality of checking for button presses for each button, but ties it to a specific object and, due to the implementation, includes the name of the button being pressed on the same line as the check happens, rather than a line apart in the uncompressed example.

By compressing the source code and abstracting some concepts, such as the button, one can also gain understanding about the rest of the program text. By showing details of practices and styles, a programmer can extrapolate from a small fragment to a larger structure. Gabriel calls this idea *locality*: it is

that characteristic of source code that enables a programmer to understand that source by looking at only a small portion of it.
(Gabriel, 1998)

In poetry, compression presents a different problem since, ultimately, the definitions of each words are not limited to the poet's own mind but also exist in the broad conceptual structures which readers hold. However, since all aspects of a program is always explicitly defined, programmers thus have the ultimate say on the definition of most of the data and functions described in code. As such, they create their own semantic contexts while, at the same time, having to take into account the context of the machine, the context of the problem, and the context(s) that their reader(s) might be coming from.

We now see that, within the same need for the appreciation of function, architecture can take opposite approaches: seeing a building as an abstract design, or as a concrete construction. In his 1951 lecture, "Building, Dwelling, Thinking", Martin Heidegger offers a perspective on these two forms of architecture. He equates top-down and bottom-up to, respectively, building as erecting, and building as cultivating. Ultimately, both of these approaches relate to human dwelling in a given location. To dwell is an engagement of thought and of action, one which leads to the construction of buildings in particular locations, arguing for a contextual adequacy of human structures to their environment⁴⁴ (Heidegger & Hofstadter, 1975). This active existence in time and space, allowing for deliberate thought and action and resulting in a better structure also equates to Gabriel's concept of habitability:

Habitability is the characteristic of source code that enables programmers coming to the code later in its life to understand its construction and intentions and to change it comfortably and confidently ...Software needs to be habitable because it always has to change ...What is important is that it be easy for programmers to come up to speed with the code, to be able to navigate

⁴⁴Speaking of a farmhouse in the Schwarzwald, he describes the chain of creation as such: "A craft which, itself sprung from dwelling, still uses its tools and frames as things, built the farmhouse.

through it effectively, to be able to understand what changes to make, and to be able to make them safely and correctly. (Gabriel, 1998)

As Heidegger returns to the etymological root of dwelling (*bauern*) in order to connect it to the possible experience of the world humans can have through language, he grounds our experience in context. His thought, between earth, man, *techne* and construction, hints at the essence that human construction—craft—as a consequence of thought and as a precedence to construction. Taking into account context and materiality, a final connection between software and architecture is actually with the field that predated, and complemented, architecture: craftsmanship.

4.3.3 Material knowledge

Architecture as a field and the architect as a role have been solidified during the Renaissance, consecrating a separation of abstract design and concrete work. This shift obfuscates the figure of the craftsman, who is relegated to the role of executioner, until the arrival of civil engineering and blueprints overwhelmingly formalized the discipline (Pevsner, 1942). While computer science, through its abstract designs, echoes the modernist architect with its pure plans, the programmer, identifying itself with the craftsman, offers different avenues for knowing artefacts.

The architect emerged from centuries of hands-on work, while the computer scientist (formerly known as mathematician in the 1940s and 1950s) was first to a whole field of practitioners 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 achievement, such as Notre Dame de Paris or the Sienna Cathedral. On the other side, letting go of structured and restricted modes of working characterizing computer pro-

gramming 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 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). Building complex structures through horizontal networks and practical knowledge is therefore possible, with consequences in terms aesthetic appreciations.

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), preferring materials and context to technological automation. Following Sennett's work on craftsmanship as a cultural practice, 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).

At the heart of knowledge transmission and acquisition of the craftsman 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), pointed at in 2.2.2, already hints at an appreciation of code beyond its formalisms and rationalisms, and towards its materiality.

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 qual-

ity (Pye, 2008; Sennett, 2009). These qualitative standards which, when pushed to their extreme, result in a craftsman's *style*, gained through practice and experience, rather than by explicit measurements (Pye, 2008)⁴⁵. Two things are concerned here, supporting the final result: tools and materials (Pye, 2008). According to Pye, a craftsman 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).

The knowledge that the craftsman derives, while being tacit (see 3.1.2), is directed at its tools, its materials, and the function ascribed to the artefact being constructed, and such knowledge is derived from a direct engagement with the first two, and a constant relation to the third. Finally, any aesthetic decoration is here to attest to the care and engagement of the individual in what is being constructed—its dwelling, in Heideggerian terms.

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 av-

⁴⁵See 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.*" (Pye, 2008)

enue 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 craftsmanship-based work in an information economy.

Beautiful code, code well-written, is 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, a different take on 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 possesses. 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, echoing Scruton's analysis that architectural aesthetics cannot be decoupled from a social sense⁴⁶. As software craftsmen assemble in loose hierarchies to construct software, the aesthetic standard is *the respect of others*, as mentioned in computer science textbooks (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, source 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

⁴⁶"it is the aesthetic sense which can transform the architect's task from the blind pursuit of an uncomprehended function into a true exercise of practical common sense." (Scruton, 2013)

with words⁴⁷.

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 possibilities for the acknowledgement of a different moral standard⁴⁸. 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

⁴⁷This disregards the impact of programming languages, the hardware they run on, and the data they process on the environment; see (Kurp, 2008)

⁴⁸Writing about resilient web development, Jeremy Keith echoes this need for material honesty: "The world of architecture has accrued its own set of design values over the years. One of those values is the principle of material honesty. One material should not be used as a substitute for another. Otherwise the end result is deceptive (Keith, 2016). "

craftsmanship 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.

To conclude this section, we have seen that architecture can offer us some heuristics when looking for aesthetic features which code can exhibit. Starting from the naïve understanding that form should follow function, we've examined how Alexander's theory of patterns, and its significant influence on the programming community⁴⁹, points not just to an explicit conditioning of form to its function (in which case we would all write hand-made Assembly code), but rather to an elusive, yet present quality, which is both problem- and context-dependent.

Along with the function of the program as an essential component of aesthetic judgment, our inquiry has also shown that program texts can present a quality that is aware of the context that the writer and reader bring with them, and of the context that it provides them, making it habitable. Software architecture and patterns are not, however, explicitly praised for their beauty, perhaps because they disregard these contexts, since they are higher-level abstractions; this implies that generic solutions are rarely elegant solutions. And yet, there is an undeniable connection between the beautiful and the universal. Departing from our investigation of software as craftsmanship, and moving through towards a more abstract discipline, we turn to scientific aesthetics.

4.4 Forms of scientific activity

As programmers learned their craft from practice and immediate engagement with their material, computer science was concomittantly develop-

⁴⁹This theory has even spawned short-lived debates about his quality without a name on stackoverflow (interstar, 2017).

ing from a seemingly more abstract discipline. Mathematicians such as Alan Turing, John Von Neumann and Grace Hopper can be seen, not just as the foreparents of the discipline of computing, but also as standing on the shoulders of a long tradition of mathematicians. Computation is one of the many branches of contemporary mathematics and, as it turns out, this discipline also has reccuringly included references to aesthetics. After the metaphors of literature and the patterned structures of architecture, we conclude our analysis of the aesthetic relation of domains contingent to source code by looking at how mathematics integrate formal presentation.

This section approaches the topic of aesthetics and mathematics in three different steps. First, we look at the objective or status of beauty in mathematics: are mathematical objects eliciting an aesthetic experience in and of themselves, or do they rely on the observer's perception? Considering the difference between abstract objects and their representation: is aesthetic representation ascribed to either, or to both? And what is the place of the observer in this judgment? Having established a particular focus on the representations of abstract objects, we then turn to the epistemic value of aesthetics, and how positive aesthetic representations in mathematics can enable insight and understanding. Finally, we complement this relation between knowledge and presentation and depart from the ends of a proof, and an evaluative appraisal of aesthetics in mathematics, by investigating the actual process of doing mathematics, concluding with topics of pedagogy and ethics.

4.4.1 Beauty in mathematics

The object of mathematics is to deal first and foremost with abstract entities, such as the circle, the number zero or the derivative, which can find their applications in fields like engineering, physics or computer science. Because of this historical separation from the practical world through

the use and development of symbols, one of the dominant discourses in the field tended to consider mathematical beauty as something intrinsic to itself, and independent from time, culture, observer, or representation itself. Indeed, a circle remains a circle in any culture, and its aesthetic properties—uniformity, symmetry—do not, at first glance, seem to be changing across time or space.

According to the Western tradition, mathematics are perhaps the first art. Aristotle, in his *Metaphysics*, wrote of beauty and mathematics as the former being most purely represented by the latter, through properties such as order, symmetry and definiteness⁵⁰. By offering insight into the harmonious arrangement of parts, it was thought that mathematics could, through beauty, provide knowledge of the nature of things, resulting in an understanding of how things generally fit together. Beauty then naturally emerges from mathematics, and mathematics can, in turn, provide an example of beauty. At this intersection, it also becomes a source of intellectual pleasure, since gaining mathematical knowledge exercises the human being's best power—that of the mind.

Arguing for this position of objective quality being revealed through beautiful manifestation, Godfrey H. Hardy writes, in his *Mathematician's Apology*, that beauty is constitutive of the objects that compose the field; their abstract quality is what removes them from the contextuality of human judgment.

A mathematician, like a painter or a poet, is a maker of patterns. If his patterns are more permanent than theirs, it is because they are made with ideas. A painter makes patterns with shapes and colours, a poet with words. ...The mathematician's patterns, like the painter's or the poet's must be beautiful; the ideas like

⁵⁰"the supreme forms of beauty are order, symmetry, and definiteness, which the mathematical sciences demonstrate in a special degree. And since these (e.g. order and definiteness) are obviously causes of many things, evidently these sciences must treat this sort of causative principles also (i.e. the beautiful) as in some sense a cause." (Aristotle, 2006)

the colours or the words, must fit together in a harmonious way.

Beauty is the first test: there is no permanent place for ugly mathematics. (Hardy, 2012)

Here, Hartman posits that it is the arrangement of ideas that possess aesthetic value, and not the arrangement of the representation of ideas. In this, he follows the position of other influential mathematicians, such as Poincaré (Poincaré, 1908), or Dirac (Kragh, 2002), who rely on beauty as a property of the mathematical object in itself. For instance, Dirac states that a physical law must necessarily stem from a beautiful mathematical theory, thus asserting that the epistemic content of the theory and its aesthetic judgment thereof are inseparable; a good mathematical theory is therefore intrinsically beautiful. Summing up these positions, Carlo Cellucci establishes proportion, order, symmetry, definiteness, harmony, unexpectedness, inevitability, economy, simplicity, specificity, and integrations as the different properties inherent to mathematical objects, as mentioned from an essentialist perspective (Cellucci, 2015). Ironically, this rather seems to hint at the multiplicity of appreciations of beauty within mathematics, with mathematicians concurring on the existence of beauty, but not agreeing on what kind of beauty pertains to mathematics. Nonetheless, they do agree that beauty is connected to understanding and epistemic acquisition. John Von Neumann, writing in 1947, states that:

One expects a mathematical theorem or a mathematical theory not only to describe and to classify in a simple and elegant way numerous and a priori disparate special cases. One also expects "elegance" in its "architectural," structural makeup. Ease in stating the problem, great difficulty in getting hold of it and in all attempts at approaching it, then again some very surprising twist by which the approach, or some part of the approach, becomes easy, etc... (Von Neumann, 1947)

The point that Von Neuman makes here is a difference between the con-

tent of the mathematical object and its structural form. Such a structural form, by organizing the connection of separate parts into a meaningful whole, makes it easy to grasp the problem. In this sense, it is both the crux of aesthetics and the crux of understanding.

Similarly, François Le Lionnais, a founding member of the Oulipo literary movement in postwar France, wrote an essay on the aesthetic of mathematics, paying attention to both the mathematical objects in and of themselves, such as e or π , but also to mathematical methods, and how they compare to traditional artistic domains such as classicism or romanticism. Without getting into the intricacies of this argumentation, we can nonetheless note that his descriptions of mathematical beauty find echoes in source code beauty. For instance, his appraisal of the proof by recurrence⁵¹ reflects similar lines of praise given by programmers to the elegance of recursive functions, which are sharing the same mathematical device (for instance, see 32 and 17 for examples of recursion as an aesthetic property). A proof by recurrence is indeed a kind of structure, which can be adapted to demonstrate different kinds of mathematical objects.

To understand is to grasp how each elements fits with others within a greater structure (either in a poem, a symphony or a theorem), with some or all of these elements being rendered sensible to the observer (Cellucci, 2015). The beauty of a mathematical object can then be ascribed in its display of the definite relation between its elements. For instance, the equation representing Euler's identity (see 4.4) demonstrates the relation between geometry, algebra and numerical analysis through a restrained set of syntactic symbols, where e is Euler's number, the base of natural logarithms, i is the imaginary unit, which by definition satisfies $i^2 = -1$, and π is

⁵¹"It seems to us that a method earns the epithet of classic when it permits the attainment of powerful effects by moderate means. A proof by recurrence is one such method. What wonderful power this procedure possesses! In one leap it can move to the end of a chain of conclusions composed of an infinite number of links, with the same ease and the same infallibility as would enter into deriving the conclusion in a trite three-part syllogism." (Le Lionnais, 1971)

$$e^{i\pi} + 1 = 0$$

Figure 4.4: Euler's identity demonstrates the relation between geometry, algebra and numerical analysis through a restrained set of syntactic symbols.

the ratio of the circumference of a circle to its diameter. Each of the symbols is necessary, definite, and establishes clear relations between each other, revealing a deep interlock of simplicity within complexity.

There is also empirical grounding for such a statement. This equation ranked first in a column in the *Mathematician Intelligencer* about the beauty of mathematical objects; the columnist, David Wells, had asked readers to rank given theorems, on a linear scale from 0 to 10, according to how beautiful they were considered (Wells, 1990)⁵². Again, while this assessment does show that there can be consensus, and thus some aspect of objectivity, in a mathematician's judgment of beauty in a mathematical object, it also showed that mathematical beauty also depends on the observer, since mathematicians provided varying accounts.

Rather than focusing on the beauty of the mathematical entities them-

⁵²Along with, for instance, the infinite prime theorem, and Fermat's "two squares" theorem.

selves, then, another perspective is to consider beauty to be found in the *representation* of mathematical , since conceptual entities can only graspable through their manifestation.

A first approach is to consider that that the beauty ascribed to mathematics and the beauty ascribed to mathematical representation are unrelated. This disjunctive view, that aesthetics and mathematics can be decoupled (e.g. there can be ugly proofs of insightful theorems, and elegant proofs of boring theorems), was first touched upon by Kant. As Starikova highlights, the philosopher operates a distinction between perceptual, disinterested beauty, and intellectual, vested beauty. Perceptual beauty, the one which can be found in the visual representations of mathematical entities, is the only beauty graspable, while intellectual beauty, that of the objects themselves, simply does not exist, "mathematics by itself being nothing but rules" (Starikova, 2018).

Such manifestation of perceptual beauty, connected to mathematical entities themselves, can nonetheless be found in the phenomenon of reproving in existing proofs, in order to make them more beautiful. Rota, for instance, associates the beauty of a piece of mathematics with the shortness of its proof, as well as with the knowledge of the existence of other, clumsier proofs⁵³ (Rota, 1997). Thus, it is not so much the content of the proof itself, nor the abstract mathematical object being proven that is the focus of aesthetic attention, but rather the process of establishing the epistemic validity of such an object.

What is useful here is technique, the demonstration of the knowledge from the prover to the observer, through the proof. As such, the assessment of aesthetics in mathematics, both as a producer and as an observer, depends on the expertise of each individual, and on the previous knowl-

⁵³"The beauty of a piece of mathematics is frequently associated with shortness of statement or of proof." and "A proof is viewed as beautiful only after one is made aware of previous, clumsier proofs." (Rota, 1997)

edge that this individual has of mathematics⁵⁴ (an assessment of the aesthetics of mathematics for non-expert is discussed in 4.4.3 below). It seems that the way that the mathematical object is presented does matter for the assessment.

If beauty is not intrinsic to the mathematical object, but rather connected to the representation of the mathematician's knowledge, there remains the question of why is beauty taken into account in the doing of mathematics. Looking at the lexical fields used by mathematicians to qualify their aesthetic experience, as reported in (Inglis & Aberdein, 2015) provides us with a clue: amongst the most used terms are "ingenious", "striking", "inspired", "creative", "beautiful", "profound" and "elegant". Some of these terms have a connection to the epistemic: for instance, something ingenious enables previously unseen connections between concepts, implying the resourcefulness and the cleverness of the originator of the idea. The next question is therefore that of the relationship between the aesthetic and the epistemic in mathematics; and in how this relation can manifest itself in source code.

4.4.2 Epistemic value of aesthetics

Caroline Jullien offers an alternative to the perception of mathematics as an autotelic aesthetic object, by retracing the definitions of beauty given by Aristotle and establishing a cognitive connection through a cross-reading of the *Metaphysics* and the *Poetics*, highlighting that "*the characteristics of beauty are thus useful properties that yield an optimal perception of the object they apply to. [...] Men can understand what is ordered, measured and delineated far better than what is chaotic, without clear boundaries, etc.*" (Jullien, 2012). She then develops this point further, building on Poincaré's

⁵⁴"Mathematical creation is not so free, hence the contrasting analogy of the landscape gardener, who needs a good grasp of the topography before getting down to creating something beautiful which needs to be based on that topography." (Thomas, 2017)

assessment of mathematical entities which fulfill aesthetic requirements and are, at the same time, an assistance towards understanding the whole of the mathematical object presented. Aesthetics, then, might not exist exclusively as intrinsic properties of a mathematical object, but rather as an epistemic device.

Her argument focuses on considering mathematics as a language of art in the Goodmanian sense of the term, investigating how mathematical notation relates to Goodman's criteria of syntactic density, semantic repleteness, semantic density, exemplification and multiple references (Jullien, 2012). She shows that, while mathematical notation might not seem to satisfy all criteria (for instance, syntactic density is only fulfilled if one takes into account graphical representations), a mathematical reasoning can present symptoms of the aesthetic, particularly through the ability to exemplify and refer to abstract entities.

However, to do that, she also includes different representations of mathematical systems, beyond typographical characters. Taking into account diagrams and graphs, it becomes easier to see how a more traditionally artistic representation of mathematics is possible. The thickness of a line, the color-coding or the spatial relationship can all express a particular class of mathematical objects; for instance, the commutative property in arithmetic can be represented in geometry through the aesthetic property of symmetry. In this work, we focus on the textual representation of source code, eluding any graph or diagram (such as the one we've seen in architectural descriptions of software systems in 4.1). Nonetheless, we have argued in 4.1 that source code qualifies as a language of art: while the syntactic repleteness does not match that of, say, painting, the unlimited typographical combinations, paired with the artificial design of programming languages as working medium enables the kinds of subtle distinctions necessary for symptoms of the aesthetic to be present.

Following Jullien, if a piece of mathematics is eliciting an aesthetic experience, or presenting positive aesthetic properties, it might then be a

support for the understanding by the viewer of this very piece of mathematics. Such a support is itself manifested in this ability to show a harmonious correspondence of parts in relation to a whole. A beautiful presentation is a cognitively encouraging presentation. The subsequent question then regards the nature of that understanding: if it does not happen as an instant stroke of enlightenment, how does it take place as a gradual process of deciphering (Rota, 1997)?

Addressing this question, Carlo Cellucci hints at the concept of fitness, meaning the appropriateness of a symbol in its denotation of a concept, and the appropriateness of concepts in their demonstration of a theorem. Only through this dual level can fitness enable understanding rather than explanation (Cellucci, 2015). This gradual conception of understanding fits the context of proofs and demonstration; when confronted with source code—that is, with the result of a thought process of one or more programmers—the processual conception of understanding seems to find its limits.

To illustrate the relation between presentation and understanding of defined conceptual entities, we can look at how the linked list, a data structure that is fundamental in computer science, can be represented in an elegant way. The linked list allows for the retrieval and manipulation of connected items, as well as for the re-arrangement of the list itself, and thus holds within it thoughtful implications in terms of organizing and accessing sequential data. To do so, each item on the list contains both its value, and the address of the next item on the list, except for the last item, which points to `null`; a last component, the `head` points to the current element of the list. A graphical representation is provided in 4.5.

The linked list implementation shown in 57 establishes a source code representation of such data structure. This comparison between a graphical representation and a textual one highlights that the graphical representation is not only artistic in the traditional sense of the term, but rather that it operates different expressive choices, and calls for attention on dif-

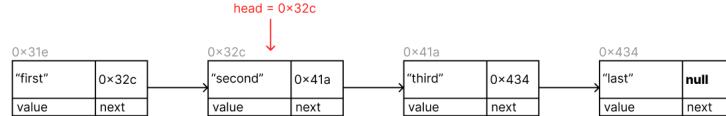


Figure 45: The linked list is an abstract data structure which acts as a fundamental conceptual entity in computer science. It is here represented as a graph, and implementations can be seen in 57 and 58.

```

struct list_item {
    int value;
    struct list_item *next;
};
typedef struct list_item list_item;

struct list {
    struct list_item *head;
};
typedef struct list list;

size_t size(list *l);
void insert_before(list *l, list_item *before, list_item *item);

```

Listing 57: A textbook example of a fundamental construct in computer science, the linked list. This header file shows all the parts which compose the concept (Kirchner, 2022a).

ferent parts of the same concept (e.g. the head of the list). On the other side, the textual representation also makes attentional choices, but to different aspects (here, the structure of the `list_item`); in this case, it seems less explanatory than the graphical representation, and limited in communicating why this is a canonical example of such a computational entity, or how did one reach this conclusion among other possible entities. The preference of graphical demonstrations over textual ones might indeed rely on the fact that our visual perception is the most developed of our system, and that our reasoning takes place through the manipulation of visual cues (Wallen, 1990).

Looking at 57, one can view the different relationships between parts

and wholes: the list item composing the list itself, the head pointer being a specific instance of the next pointer, and the different methods to access or modify the list itself. Seeing all of these together suggest an understanding of the whole through the parts which is nowhere explicitly described but everywhere suggested. On the contrary, the diagram provided in 4.5 provides a much more immediate understanding of how the linked list is structured. As such, its aesthetic properties (spacing, weight, color) contribute to highlighting the smae parts as defined in 57.

Rather than in the static description of the structure, we can look at the operations which can be performed on it in order to suggest implicit qualities of the object at hand. The linked list example (see 57) might be considered aesthetically pleasing only at a particular level of skill. However, once we start manipulate the concept, we can grasp its underlying properties. In 58, we have reproduced two functions which perform the same operation: given a list and an element, they remove the element from the list.

The distinction is made clear via the function names between a beginner level (`remove_cs101`, labelled "CS101" for the course number of introduction to computer science) and an expert level (`remove_elegant`). In the first one, we see two main statements, `while` and `if`. The first statement looks for the element to be removed by looping over the list. Once it has found it, it hands it over the next statement, which checks for the particular edge case where the target element might be the first one, which needs to be handled differently than for all other cases. In this case, the representation of this operation does not quite reach into the generic.

The second function is more complex, yet more enlightening. In order to get rid of the particular edge case which does not symbolize the universality of such a procedure, the author⁵⁵ makes a heavy use of the pointer notation, which allows a program to refer to either the content of a variable, or to the address at which the content is stored.

⁵⁵The author of this particular elegant re-write is Linus Torvalds, orinal author of the Linux kernel (Torvalds, 2016).

```

void remove_cs101(list *l, list_item *target)
{
    list_item *cur = l->head, *prev = NULL;
    while (cur != target)
    {
        prev = cur;
        cur = cur->next;
    }
    if (prev)
        prev->next = cur->next;
    else
        l->head = cur->next;
}

void remove_elegant(list *l, list_item *target)
{
    list_item **p = &l->head;
    while (*p != target)
        p = &(*p)->next;
    *p = target->next;
}

```

Listing 58: A comparison of how to remove an element from a list, with elegance depending on the skill level of the author (Kirchner, 2022b).

This use of pointers implies a change in the mental model when considering a linked list. While the first implementation operates on `list_item` elements, and then individually deals with the sub parts (such as the `next` field), the second implementation considers the list mostly as a series of pointers, thus reducing the conceptual overload, and increasing the functional efficiency at the price of initially more cryptic notation.

Subtle notational changes can therefore flip the representation of a conceptual entity. Rather than being separated from purpose (in the case of mathematics, that function being proving a theorem), aesthetics is integrated into it by facilitating the understanding of the connection between, and the reasoning for mental or computational operations. For instance, writing about elegant mathematical proofs, John Barker argues that aesthetics are involved in implicit understanding:

Grasping a proof, understanding its gist, seeing why it works, is an important further step, and an essential step if one is to be-

come a competent mathematician. However, by simply following each move in a proof, one has learned everything that is explicitly stated in the proof. Therefore, in really understanding a proof, one must be learning something that is not explicitly stated in it.

(Barker, 2009)

Still, whether an aesthetic judgment relies on perceived qualities, or if it only relies on the quality of an idea, is still up for debate. For instance, Starikova that "[A]lthough visual representations are involved and understanding does rely on them, it is clearly non-perceptual beauty that initiates aesthetic judgment" (Starikova, 2018), pointing back to the distinction above as to whether beauty is perceived as intrinsic to the mathematical object, or intrinsic to its representation.

Here, we argue that, when it comes to source code, adequate representation of the idea is necessary in order to elicit an aesthetic experience, following our conception of understanding through an embodied lens. However, aesthetic judgments also depend on the nature of background knowledge that the reader holds when engaging with a program text. As we saw in 58, a beginner might appreciate the conceptual beauty of the linked list, while an expert might appreciate the beautiful representation of the linked list.

On one side, the lack of pre-existing knowledge involves the deciphering of symbols and thus immediate attention to form. On the other side, the pre-existence of knowledge allows one to focus on the quality and details of the presentation, such as when mathematicians decide to find more beautiful proofs to an existing theorem. In this case, the knowledge of the theorem, and how its intellectually-perceived simplicity can be translated into a sensually-perceived simplicity and an aesthetic judgment on the form. Here, the aesthetic judgment precedes the intellectual judgment, all the while not guaranteeing a positive intellectual judgment (e.g. the abstract object, whether program function or mathematical theorem, is

presented in an aesthetically-pleasing manner, but remains shallow, boring, non-sensical or wrong).

We argue here that both intellectual pleasure and aesthetic pleasure happen in a dialogic fashion, considering the symbols and the meanings reciprocally, until intellectual and aesthetic judgments have been given. This is in line with Rota's critique of the term "enlightenment" or "insight" in his phenomenological account of beauty in mathematics. The process of discovery and understanding is a much longer one than a simple stroke of genius experienced by the receiver (Rota, 1997).

An aesthetic experience in mathematics involves uncovering the connections between aesthetic and epistemic value being represented through a mathematical symbol system. However, such a conception seems to take place as a gradual process of discovery, both from the writer and from the reader, rather than intrinsic aesthetic value existing in a given mathematical concept. Seen in the light of skill-based aesthetic judgment, this chronological unfolding points towards a final aspect of aesthetics in mathematics specifically, and in the sciences in general: aesthetics as heuristics for knowledge acquisition.

4.4.3 Aesthetics as heuristics

So far, we had been looking at how aesthetics are evaluated in a finished state—that is, once the form of the object (whether a proof or a program text) has stabilized. In doing so, we have left aside a significant aspect of the matter. Aesthetics in mathematics do not need to be seen exclusively as an end, but also as a mean, as a part of the cognitive process engaged to achieve a result. As such, we will see how they also serve as a useful heuristic, both from a personal and social perspectives. Since the ultimate purpose of mathematics specifically, and scientific activities in general, is the establishment of truths, one can only follow that beauty has but a secondary role to play—though that is not to say superfluous.

Complementing the opinions of mathematicians at the turn of the 20th century, Nathalie Sinclair offers a typology of the multiple roles that beauty plays in mathematics. Beyond the one that we have investigated in the previous sections, which she calls the *evaluative* role of beauty, in determining the epistemic value of a conceptual object, she also proposes to look at a *generative* role and at a *motivational* role (Sinclair, 2011). The latter helps the mathematician direct their attention to worthy problems—something which is of limited equivalence in source code, since programming mostly derives from external constraints. The former holds a guiding role during the inquiry itself, once the domain of inquiry has been chosen. It helps in generating new ideas and insights as one works through a problem. This aesthetic sense can be productive both in its positive evaluations—implying one might be treading a fruitful path—as well as negative—hinting that something might not be conceptually well-formed because it is not formally well-presented⁵⁶. According to Root-Bernstein, the informal insights of aesthetic intuition precede formal logic. Only when we explicitly recognize that the “tools of thinking” and the “tools of communication” are distinct can we understand the intimate, yet tenuous, connection between thought and language, imagination and logic (Root-Bernstein, 2002).

This is echoed in Norbert Wiener’s perception of aesthetics in mathematics as a way to structure a knowledge that is still in the process of being formed, in order to optimize short-term memory as the mental model of the conceptual object being grasped is still being built⁵⁷. This descrip-

⁵⁶“The realization that we recognize problems through our anti-aesthetic response to them provides an important clue as to how we go about defining the nature of the problem and recognize its solution. The nature of the disjunction between our aesthetic expectations and what we observe or think we know reveals the detailed characteristics of the specific problem that presents itself.” (Root-Bernstein, 2002)

⁵⁷“The mathematician’s power to operate with temporary emotional symbols and to organize out of them a semipermanent, recallable language. If one is not able to do this, one is likely to find that his ideas evaporate due to the sheer difficulty of preserving them in an as yet unfor-

tion of a sort of landmark item, in the geographical sense, echoes the role of beacons described by Détienne (Detienne, 2001) and mentioned in 3.2.3. One can therefore consider an aesthetically pleasing element to serve as a sort of beacon used by programmers to construct a mental representation of the program text they are reading or writing.

A representation does not need to be of an effective proof in order to be nonetheless considered functional. A sketch of a concept might even evoke more in certain readers than a fully detailed implementation, offering a direction into which further fruitful inquiry.

For instance, the listing 59 shows such a sketch, as it represents the essential components of a regular expression matcher, as featured in the *Beautiful Code* edited volume. Regular expressions are a form of linguistic formal pattern that serve as an input to a regular expression matcher in order to find particular patterns of text in an input string. Given a input text file, a regular expression matcher could find a pattern such as "*any consecutive list of characters, starting with any number of alphanumeric characters, followed by a dot, followed by at least one character and at most seven characters*"—in essence a rough description of a file name and extension. Building such a system is not a trivial endeavour.

In this case, the essential components of the matcher are implemented, in a clear and concise way. It highlights the overall structure (a general `match` function, with `matchhere` and `matchstar` handling separate cases), the process of looping over an input string, the fundamentals of handling different patterns, and within those the fundamentals of handling different characters in relation to the current pattern. Each part is clearly delineated (and fit for its separate purposes) and contributes to an understanding of the whole, by limiting itself to displaying its essence.

It must be pointed out here that the regular expression is functional at its core: in less than 50 lines, it performs the core operations of the system it represents, while a fully-functional implementation, such as the one in

mulated shape." quoted by Sinclair in (Sinclair, 2004)

```

/* match: search for regexp anywhere in text */
int match(char *regexp, char *text)
{
    if (regexp[0] == '^')
        return matchhere(regexp + 1, text);
    do
    { /* must look even if string is empty */
        if (matchhere(regexp, text))
            return 1;
    } while (*text++ != '\0');
    return 0;
}

/* matchhere: search for regexp at beginning of text */
int matchhere(char *regexp, char *text)
{
    if (regexp[0] == '\0')
        return 1;
    if (regexp[1] == '*')
        return matchstar(regexp[0], regexp + 2, text);

    if (regexp[0] == '$' && regexp[1] == '\0')
        return *text == '\0';
    if (*text != '\0' && (regexp[0] == '.' || regexp[0] == *text))
        return matchhere(regexp + 1, text + 1);
    return 0;
}

/* matchstar: search for c*regexp at beginning of text */
int matchstar(int c, char *regexp, char *text)
{
    do
    { /* a * matches zero or more instances */
        if (matchhere(regexp, text))
            return 1;
    } while (*text != '\0' && (*text++ == c || c == '.'));
    return 0;
}

```

Listing 59: A regular expression matcher by Rob Pike, praised for its elegance and conciseness, but not for its practical utility (Oram & Wilson, 2007)

Python's `re` module, is more than 2000 lines (Secret Labs AB, 2001). The beauty found by Brian Kernighan in this program text is that the core of the idea is represented elegantly, while leaving avenues for exploration to the reader⁵⁸.

Mathematics, like source code, therefore pay close attention to how formal presentation facilitates the cognitive grasping of abstract concepts. Reducing and organizing literal tokens into conceptually coherent units, and meaningful relations to other units—for instance, having the code in 59 reversed, with the `match()` function at the bottom of the document would represent a different level of importance of that entrypoint function, which would complicate the understanding of how the source code functions.

One of the virtues of the listing is that it is particularly beneficial to students, helping them grasp the important parts without being overloaded with too many technical details. Nathalie Sinclair further develops the importance of aesthetically pleasing textual objects representing mathematical concepts in order to facilitate learning. She positions her argument as a response to the strict focus of the studies in mathematics on the perceptions and reports of highly successful individuals. If individuals like Poincaré, Hardy and Dirac can self-report their experiences, she inquires into the ability for individuals of a different skill level to experience generative aesthetic experiences, experiences where the encountering of an aesthetic symptom generates new directions for ideas. In a subsequent work, she describes the perception of mathematics students as such:

The aesthetic capacity of the student relates to her sensibility in combining information and imagination when making purposeful decisions, regarding meaning and pleasure. (Sinclair, 2011)

Emotion and intellect are no longer antitheses, and can be reported by

⁵⁸Brian Kernighan concludes his analysis of this piece of code as such: "I don't know of another piece of code that does so much in so few lines while providing such a rich source of insight and further ideas." (Kernighan, 2007)

$$\begin{aligned}
 (4.1) \quad & \sqrt{2} = p/q \\
 (4.2) \quad & \sqrt{2} * q = p \\
 (4.3) \quad & p = \sqrt{2} * q \\
 (4.4) \quad & (\sqrt{2})^2 = (p/q)^2 \\
 (4.5) \quad & 2 = p^2/q^2 \\
 (4.6) \quad & p^2 = 2 * q^2
 \end{aligned}$$

Figure 4.6: Steps of transformation to approach an epistemic value in finding whether or not the square root of 2 is an rational number.

students as well. From her investigations, then, it seems that the heuristic value works across skill levels, whether one holds a Fields medal or a high-school degree. Doing similar work, Seymour Papert aimed at evaluating the functional role of aesthetics by documenting a group of non-experts working through a proof that the square root of 2 is an irrational number. After a series of transformative steps, the subjects of the study manged to eliminate the square root symbol by elevating the two other variables to the power of two⁵⁹, as in 4.6.

Papert conceptualizes such an observation as a phase of play, a phase of playing which is aesthetic insofar as the person doing mathematics is delimitating an area of exploration, qualitatively trying to fit things together, and seeking patterns that connect or integrate (Papert, 1978), and thus looking to identify parts which would seem to fit a yet-to-be-determined whole. This also seems to confirm the perspective that there are some structures that are meaningful to the mathematician.

An interesting aspect of this conception of aesthetics by both Papert and Sinclair is their temporal component. While, for evaluative aesthetics,

⁵⁹Incidentally, this process of elevating to a power to get rid of a square root is the same heuristic used in the highly-optimized piece of hacker code calculating the inverse square root of a number, listed in 13 and discussed in 2.1.2

one can grasp the formal representation of the mathematical object in one sweep, this generative role hints at a more important need to develop over time. This opens up a new similarity with source code, by shifting from the reader to the writer. On one side, Sinclair connects this unfolding over time with Dewey's theory of inquiry and with Polanyi's personal knowledge theory, connecting further the psychological perception with the role of aesthetics. Both Dewey and Polanyi offer a conception of knowledge creation which relies particularly on a step-by step development rather than immediate enlightenment (Polanyi & Grene, 1969; Sinclair, 2004); it is precisely this distinction which Papert addresses with his comparison of aesthetics as *gestalt* (evaluative) or *sequential* (generative).

Taking from Dewey's proposal of what an aesthetic experience is⁶⁰, we can connect it back to a sequential aesthetic perception in Papert's term, one of learning and discovery, but also to the practice of writing good source code.

In programming practice, the process of working through the establishment of a valuable epistemic object through the sequential change of representations is called *refactoring*. As described by Martin Fowler, author of an eponymous book, refactoring consists in improving the textual design of an existing program text, while retaining an identical function. The crux of the process lies in applying a series of small syntactical transformations, each of which help to sharpen the fitness of the parts to which these transformations are applied. Ultimately, the cumulative effect of each of these syntactical transformations ends up being significant in terms of program comprehension, bringing it closer to a sense of elegance (Fowler et al., 1999)⁶¹.

⁶⁰Dewey presents it as having first and foremost a temporal structure, something that is dynamic, because it takes a certain time to complete, time to overcome obstacles and accumulate sense perceptions and knowledge, following a certain direction, a teleology hopefully concluding in a certain sense of pleasure and fulfillment. (Leddy & Puolakka, 2021)

⁶¹We have described an instance of this process in 4.3.2, with a starting point in 2, and a conclusion in 3

Finally, extending from this personal and psychological perspective on the development of epistemic value through the pursuit of aesthetic perceptions, we can note a final dimension to aesthetics in mathematics: a social component. Shifting our attention away from the modes of mathematical inquiry of individual mathematician, Sinclair and Primm have highlighted the practices of the community as a whole, including how truths are named, manipulated and negotiated. (Sinclair, 2011).

On one side, this amounts to uncovering the fact that scientific problems are being decided upon and researched based on particular values and conventions, conventions which then trickle down into the presentation of results, highlighting trends and social formations both in terms of content of research and style of research (Depaz, 2023). The interpretation provided by Pimm and Sinclair is that aesthetics, through "good taste" subtly reify a power relationship and exclude practitioners by delimiting what is proper writing and proper research (Sinclair & Pimm, 2010).

While one could argue for a similar power dynamic when it comes to programming style, one notable difference we see with programming is the highly interactive collaborative environment in which the productive work can be done. Particularly in the case of software engineering, the fact that a given program text is being worked on by different individuals of different skill levels and at different moments also suggests a social function of aesthetics as a means to harmonize social processes. The evaluative posture of the reader in giving a positive value judgment on a given program text or mathematical proof also implies that this positive judgment was given as a generative role; that is, the aesthetic symptoms are made visible by a writer in search of elegant function and epistemic communication, and appreciated by the reader as an indicator of a work well-done (Tomov, 2016).

This implies a certain sense of care that was being given to the program text, or to the mathematical proof, which in turn suggests a certain functional quality in the finished object. Beautiful mathematics, as beautiful

code, can therefore be seen as a sign that someone cared for others to understand it clearly.

Aesthetics, then, complement more traditional notions of scientific thinking, from representing a mathematical object, enabling access to the conceptual nature and implications of this object, as well as providing useful heuristics in establishing a new object. What remains, and what will be taken up in the next chapter, is to *"reify this meta-logic as a set of rules, axioms, or practices."* (Root-Bernstein, 2002), by establishing which mathematical approaches fit with source code aesthetics.

In this chapter, we have established a more thorough connection between aesthetics and cognition. First at the philosophical level, we established how source code fits within Nelson Goodman's conception of what is a language of art, before complementing this ability for an aesthetic experience to communicate complex concepts with more contemporary research, including contribution from cognitive sciences.

We then moved to more specific domains, examining both how their aesthetic properties engage with cognition, but also how these might relate to those held by source code. Looking at literature, we paid attention to how metaphors, embodied cognition and spatial representations are all devices allowing for the evocation of complex world spaces and cultural references, facilitating the comprehension of (electronic) poetry and prose.

Turning to architecture, we acknowledged the role of function in the conception of modernist aesthetics, one which focuses on the plan rather than on the building, before contrasting this approach with the theories of Christopher Alexander. His concepts of patterns and habitability have been widely transposed in programming practice, highlighting a tension between top-down, abstract design, with bottom-up, hands-on engagement. This notion of direct material engagement led us to further examine

how craft folds ties to architecture, and how it facilitates a particular kind of knowledge production and value judgment.

Finally, turning to mathematics, we distinguished two main approaches: an evaluative aesthetics, where the representation of a mathematical object has an epistemic function, and a generative aesthetics, which works as a heuristic from a writer's perspective, and often remains unseen to the reader, as it is presented in its final form, without the multiple steps of formal transformations that led to the result.

Throughout, we compared how these specific aesthetic approaches related to source code. Since source code is presented by programmers as existing along these domains of practice, this has allowed us to further refine a specific aesthetics of source code. The next chapter brings the concepts identified in these domains into a dialogue in order to constitute a coherent view. To do so, we will start from source code's material: programming languages.

Chapter 5

Machine languages

After analyzing the discourses of programmers on beautiful code, after highlighting the specific cognitive complexities inherent to software and how they are dealt with, and after having investigated how aesthetics enable various forms of understanding in adjacent fields, we now offer a framework for the aesthetics of source code.

To do this, this chapter begins with the medium of source code: programming languages. Understanding what they are and how they are used will allow us to highlight two important aspects. First, that there is a tension between human-meaning and machine-meaning, a tension between different interpretations of the same syntax. Second, it will allow us to highlight another contextual aspect of source code aesthetics—just like natural languages, machine languages also act as linguistic communities.

Once we laid this material groundwork, we propose two approaches to the aesthetic manifestations in program texts. First, we build on a close-reading approach to suggest a framework composed of various scales. Focusing on the spatiality of program texts, we will show how programming languages act as an interface between a program text and a mental model. We then develop on how syntax and vocabulary make use of metaphors to enable the representation of positive values such as abstraction, openness

and function.

It is with the concept of function that we conclude the chapter, and with the essential role that function plays in aesthetic appreciation. That is, we will show that such role is dual: a functional source code is required for aesthetic judgment to take place, and the aesthetic experience has the function of enabling understanding.

5.1 Linguistic interfaces

Software is an idea ultimately represented in specific hardware configurations. The immediate medium of this representation, from the programmer's perspective, is the programming language in which the idea is written down. Programming languages have so far been set aside when examining which sensual aspects of source code resulted in what could be deemed a "beautiful" program text. And yet, the relationship between semantics (deep-structure) and its syntactic representation (surface-structure) is framed by programming languages, as they define the legal organization of form.

This section examines the influence of programming languages on the aesthetic manifestations of source code. To do so, we first go over a broad description of programming languages, focusing on what makes a programming language expressive. Second, we touch upon the problem of semantics in programming languages, and how they might differ from a human understanding of semantics. We then we assess their fit as an artistic, expressive system by introducing notions to style and idiomacity in programming language communities. In so doing, we highlight a couple of computing-specific concepts that are made accessible by programming languages, discussing how different linguistic interfaces propose different representations.

5.1.1 Programming languages

We start by recalling the historical and technical developments of programming languages, relocating them as an interface between hardware and software. With a better technical understanding, this will allow us to pinpoint the overlap and differences between human semantics and machine semantics.

History and developments

A programming language is a strictly-defined set of syntactic rules and symbols for describing instructions to be executed by the processor. The history of programming languages is, in a sense, the history of decoupling the means of creating software from hardware. The earliest programming languages were embedded in hardware itself, such as piano rolls and punched cards for Jacquard looms (Sack, 2019). Operating on similar principles, the first electric computers—such as the ENIAC, the UNIVAC or the MUC—still required manual re-wiring in order to implement any change in the algorithm being computed. This process then gave way to programming through the stack of cards fed into the machine, a more modular process which nonetheless retained a definite material aspect. It is with the shift to the stored-program model, at the dawn of the 1950s, that the programs could be written, stored, recalled and executed in their electro-(mecha)nical form, essentially freeing the software result from any immediately physical representation.

This tendency to have software gradually separate from hardware saw a parallel in the development of programming languages themselves. Ultimately, any software instruction needs to execute one of the built-in, hard-wired instructions of the processor. Also called *machine language*, these instructions set describe the specific implementation of the most common operations executed by a computer (e.g. `add`, `move`, `read`, `load`, etc.), and are part of the oldest and most direct semantic interface to the hardware.

These operations are ultimately represented as binary numbers to the processing unit. To represent these binary combinations, a first layer of a family of languages called Assembly, provides a syntax which is loosely based on English. When read by the CPU, each of these Assembly mnemonics is converted into binary representation¹. Considered today as some of the most low-level code one can write, Assembly languages are machine-dependent, featuring a one-to-one translation from English keywords to the kind of instruction sets known to the processor they are expected to interface with. As such, a program written for a particular architecture of a computer (e.g. x86 or ARM) cannot be executed without any modifications on another machine.

The first widely acknowledged high-level language which allowed for a complete decoupling of hardware and software is FORTRAN². At this point, programmers did not need to care about the specifics of the machine that they were running on anymore, and found more freedom in their exploration of what could be done in writing software, expanding beyond scientific and military applications into the commercial world (see 2.1). Moving away from hand-crafted and platform-specific Assembly code also implied a certain sense of looseness incompatible with the extension of its application domain: widening the problem domain demanded tightening the specification of such languages. As such, FORTRAN³, and the subsequent COBOL, Lisp and ALGOL 58 also started being concerned with the specific definition of their syntax in a non-ambiguous manner to ensure reliability. Using Backus-Naur Form notation, it became possible to formalize their syntactic rules in order to prevent any unexpected behaviour and support rigorous reasoning for the implementation and research of current and subsequent languages. With such specifications, and with the decoupling

¹For an example of Assembly language translated into machine code, see 41 and 42

²Even though programming languages such as Plankalkül, Short Code and Autocode were partial proposals of such decoupling before FORTRAN.

³An acronym for FORMula TRANslator, thus making clear its role as a mediator.

from hardware, programming languages became, in a way, context-free.

The context-free grammatical basis for programming allowed for the further development of compilers and interpreters, binary programs which, given a syntactically-valid program text, output their machine code representation. Such a machine-code representation can then be executed by the processor⁴. At this point, a defining aspect of programming languages is their theoretical lack of ambiguity. This need for disambiguation was reflected both in the engineering roots of computation⁵ and in their formal mathematic roots notation⁶, and was thus a requirement of the further development of functional software engineering.

Nowadays, most programming languages are Turing-complete: that is, their design allows for the implementation of a Turing machine and therefore for the simulation of any possible aspect of computation. This means that any programming language that is Turing-complete is *functionally* equivalent to any other Turing-complete programming language, creating essentially a chain of equivalency between all programming languages. And yet, programming language history is full of rise and fall of languages, of hypes and dissapointments, of self-claimed beautiful ones and criticized ugly ones, from COBOL to Ada, Delphi and C. This is because, given such a wide, quasi-universal problem set, the decision space requires creative constraints: individual programmers resort to different approaches of writing computational procedures, echoing what Gilles Gaston-Granger undestands as *style*, as a formal way to approach the production and communication of aesthetic, linguistic and scientific works (Granger, 1988). We have already seen one example of such difference in approaching the domain of computation: compilation vs. interpretation. While the input

⁴The main difference between a compiler and an interpreter is that the compiler parses the whole program text as once, resulting in a binary object, while interpreters parse only one line at a time, which is then immediately executed.

⁵Punch cards and electrical circuits are ultimately discrete—hole or no hole, voltage or no voltage.

⁶For instance, Plankalkül was based on Frege's *Begriffschrift*, a lineage we've seen in 3.1.2

and outputs are the same⁷, there are pros and cons⁸ to each approach, which in turn allows programmers to bestow value judgments on which one they consider better than the other. Ultimately all programming languages need to address these basic components of computation, but they can do it in the way they want. Such basic components are, according to Milner (Milner, 1996):

- *data*: what kinds of basic datatypes are built-in the language, e.g. signed integers, classes
- *primitive operations*: how can the programmer directly operate on data, e.g. boolean logic, assignments, arithmetic operations
- *sequence control*: how the flow of the program can be manipulated and constrained, e.g. if, while statements
- *data control*: how the data can be initialized and assigned, e.g. type-safe vs. type-unsafe
- *storage management*: how the programming language handles input/output pipelines
- *operating environment*: how the program can run, e.g. virtual machine or not

This decision to change the way of doing something while retaining the same goal is particularly salient in the emergence of programming paradigms. A programming paradigm is an approach to programming based on a coherent set of principles, sometimes involving mathematical theory or a specific domain of application. Some of these concepts include

⁷a program text goes in, and machine code comes out

⁸For instance, a compiled binary does not need an extra runtime to be executed on a machine, but cannot be immediately used on a different architecture than the one it was compiled for.

encapsulation and interfaces (in object-oriented programming), pure function and lacks of side effects (in functional programming), or mathematical logic (in declarative programming). Each paradigm supports a set of concepts that makes it the best for a certain kind of problem (Van Roy, 2012), these concepts in turn act as stances which influence how to approach, represent and prioritize the computational concepts mentioned above, and as tools to operate on their problem domain.

Along with programming paradigms, programming languages also present syntactic affordances for engaging with computational concepts. Nonetheless, this is only one part of the picture: the interpretation of syntax necessarily involves semantics. Machine semantics, as we will see, operate a delicate balance between computational operations and human assumptions.

Machine semantics and human semantics

One of the reasonings behind the formal approach to programming languages is, according to the designers of ALGOL 58, the dissatisfaction with the fact that subtle semantic questions remained unanswered due to a lack of clear description (Sethi, 1996). If the goal of a program text is to produce a functional and deterministic execution, then programming languages must be syntactically unambiguous, and the compiler must be given a framework to interpret this syntax. The very requirement for semantic representation in program language design is first and foremost due to the fact that:

The first and most obvious point is that whenever someone writes a program, it is a program about something. (Winograd, 1982)

The issue that he points out in the rest of his work is that humans and computers do not have the same understanding of what a program text is about. In general, semantics have the properties of aboutness and directedness (they point towards something external to them), and syntax has

the property of (local) consistency and combination (they function as a mostly closed system). Looking at programming languages as applied mathematics, in the sense that it is the art and science of constituting complex systems through the manipulation formal tokens, tokens which in turn represent elements in the world of some kind, we arrive at the issue of defining semantics in strictly computer-understandable terms.

In attempting to develop early forms of artificial intelligence in the 1970s, Terry Winograd and Fernando Flores develop a framework for machine cognition as related to human cognition, through the analysis of language-based meaning-making (Winograd & Flores, 1986). In short, they consider meaning as created by a process of active reading, in which the linguistic form enables interpretation, rather than exclusively conveying information. They further state that interpretation happens through *grounding*, essentially contextualizing information in order to interpret it and extract meaning. He identifies three different kinds of grounding: experiential, formal, and social. The *experiential* grounding, in which verification is made by direct observation, relates to the role of the senses in the constitution of the conceptual structures that enable our understanding of the world—also known as the material implementation of knowledge. The *formal* grounding relies on logical and logical statements to deduce meaning from previous, given statements that are known, which we can see at play in mathematical reasoning. Finally, *social* grounding relies on a community of individuals sharing similar conceptual structures in order to qualify for meaning to be confirmed. Of these three groundings, programming languages rely on the second.

The reason for the bypassing of experiential and social grounding can be found in one of the foundations of computer science, as well as information science: Claude Shannon's mathematical theory of communication. In it, he postulates the separation of meaning from information, making only the distinction between signal and noise. Only formal manipulation

of signal can then reconstitute meaning⁹. We think of computers as digital machines but they can also be seen as only the digital implementation of the phenomenon of computation. Indeed, according to Brian Cantwell Smith, computing is *meaning mechanically realized*, due to the fact that the discipline has both mechanical and non-mechanical lineages(Smith, 2016). It is therefore through formal logic that one can recreate meaning through the exclusive use of the computer.

This machine meaning is also represented through several layers. A computer is a collection of layers, each defining different levels of machines, with different semantic capabilities. First, it is a physical machine, dealing with voltage differences. These voltage differences are then quantized into binary symbols, in order to become manipulable by a logical machine. From this logical machine is built an abstract machine, which uses logical grounding in order to execute specific, pre-determined commands. The interpretation of which commands to execute, however, leaves no room for the kind of semantic room for error that humans exhibit (particularly in hermeneutics). It is a strictly defined mapping of an input to an output, whose first manifestation can be found in the symbols table in Turing's seminal paper (Turing, 1936). The abstract machine, in turn, allows for high-level machines (or, more precisely, high-level languages which can implement any other abstract machine). These languages themselves have linguistic constructs which allow the development of representational schemes for data (i.e. data structures such as structs, lists, tuples, objects, etc.). Finally, the last frontier, so to speak, is the problem domain: the thing(s) that the programmer is talking about and intends to act upon. Going back down the ladder of abstractions, these entities in the problem domain are then represented in data structures, manipulated through high-level languages, processed by an abstract machine and executed by a logical machine which turns these pre-established commands

⁹An affordance that is shared in distinguishing literature from gibberish, according to Peter Suber(Suber, 1988)

into voltage variations.

The problem domain is akin to a semantic domain, a set of related meaningful entities, operating within a specific context, and which a particular syntax refers to. Yet, there is only one context which the computer provides: itself. Within this unique context, semantics still hold a place in any programming language textbook, and is addressed regularly in programming language research. Concretely, *semantics in computer programming focuses on how variables and functions should behave in relation to one another* (Sethi, 1996). Given the statement $l := j + p$, the goal of programming language semantics is to deduce what is the correct way to process such a statement; there will be different ways to do so depending on the value and the type of the j and p variables. If they are strings, then the value of j will be their concatenation, putting one next to the other. If they are numbers, it will be their addition, and so on.

This problem of determining which operation should take place given a particular type of variables requires the reconciliation of the name of entities, tokens in source code, with the entities themselves, composed of a value and a type. The way this is achieved is actually quite similar to how syntax is dealt with. The compiler (or interpreter), after lexical analysis, constructs an abstract syntax tree (AST) representation of the statement, separating it, in the above case, in the tokens: l , $:=$, j , $+$ and p . Among these, $:=$ and $+$ are considered terminal nodes, or leaves, while the other values still need to be determined. The second pass represents a second abstract syntax tree through a so-called semantic analysis, which then *decorates* the first tree, assigning specific values (attributes) and types to the non-terminal nodes, given the working environment (e.g. production, development, test). This process is called *binding*, as it associates (binds) the name of a variable with its value and its type.

Semantics is thus the decoration of parsed ASTs, evaluating attribute—which can be either synthesized or inherited. Since decoration is the addition of a new layer (a semantic layer) on top of a base layer (a syntactic one),

but of a similar tree form, this leads to the use of what can be described as a *meta-syntax tree*.

Regarding when the values are being bound, there are multiple different binding times, such as language-design time (when the meaning of `+` is defined), compile time, linker time, and program-writing time. It is only during the last one of these times, that the programmer inserts their own interpretation of a particular meaning (e.g. `j := "jouer"`, meaning one of the four possible actions to be taken from the start screen of a hypothetical video game). Such a specific meaning is then shadowed by its literal representation (the five consecutive characters which form the string) and its pre-defined type (here, it would be the `string` type, although different languages have different terms to refer to the same consecutive list of alphanumeric characters).

Ultimately, this process shows that the meaning of a formal expression can, with significant difficulty and clumsiness, nonetheless be explained; but the conceptual content still eludes the computer, varying from the mundane (e.g. a simple counter) to the almost-esoteric (e.g. a playful activity). Even the most human-beautiful code cannot force the computer to deal with new environments in which meaning has, imperceptibly, changed. Indeed,

In programming languages, variables are truly variable, whereas variables in mathematics are actually constant (Wirth, 2003).

This implies that the content of the variables, when set during program-writing time, might throw off the whole interpretative process of the computer. In turn, this would transform a functional program into a buggy one, defeating the very purpose of the program. While programming languages are rigorously specified, they are nonetheless designed in a way that leaves space for the programmer's expressivity.

At this point, the only thing that the computer does know that the programmer does not is how the code is represented in an AST, and where

in physical memory is located the data required to give meaning to that tree(Stansifer, 1994). We might hypothesize that beautiful code, from the computer's perspective, is code which is tailored to its physical architecture, a feat which might only be realistically available when writing in Assembly, with deep knowledge of the hardware architecture being worked on.

Just like some human concepts that are complicated to make the computer on its own terms, there are also computer concepts that are hard to grasp for humans. As we've seen with software patterns, what also matters to programming languages is not just their design, but their *situated use*:

It must be a pleasure and a joy to work with a language, at least for the orderly mind. The language is the primary, daily tool. If the programmer cannot love his tool, he cannot love his work, and he cannot identify himself with it. (Wirth, 2003)

While there is only one version of how the computer interprets instructions, it is through programming languages that both form and content, syntax and semantics are made accessible to the programmer. Within computation as a whole, a plethora of programming languages exist, designed by humans for humans, differentiating themselves by how the representations they afford guide the programmer in reading and writing source code.

5.1.2 Qualities of programming languages

All programming languages stem from and relate to a single commonality—Turing-completeness and data processing—, and yet these linguistic interfaces nonetheless offer many approaches to performing computation, including a diversity and reliability of functional affordances and stylistic phrasing. Since diversity within equivalence supports qualified preference, we can now examine what makes a pro-

gramming language good—i.e. receive a positive value judgment—before turning to the question of the extent to which a good programming language enables the writing of good program texts.

Every programming language of practical use takes a particular approach to those basic components, sometimes backed by an extended rationale (e.g. ALGOL 68), or sometimes not (e.g. JavaScript). In the case in which one is circumscribed to context-free grammars, it would be possible to optimize a particular language for a quantifiable standard (e.g. compile time, time use, cycles used). And still, as computers exist to solve problems beyond their own technical specifications, such problems are diverse in nature and therefore necessitate different approaches¹⁰. These different approaches to the problem domain are in turn influenced the development of different programming languages and paradigms, since a problem domain might have different data representations (e.g. objects, text strings, formal rules, dynamic models, etc.) or data flows (e.g. sequential, parallel, non-deterministic). For instance, two of the early programming languages, FORTRAN and Lisp, addressed two very different problem domains: the accounting needs of businesses and the development of formal rules for artificial intelligence, respectively. Within programming languages, there is room to distinguish better ones and worse ones, based on particular qualities, and given standards.

What makes a good programming language is a matter which has been discussed amongst computer scientists, at least since the GOTO statement has been publicly considered harmful (Dijkstra, 1968), or that the BASIC language is damaging to one's cognitive abilities¹¹. Some of these discussions include both subjective arguments over preferred languages, as well as objective arguments related to performance and ease-of-use (Gannon &

¹⁰Patterns, addressed in 4.3.2 are one way that diverse approaches can be applied to diverse problems

¹¹"It is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration." (Dijkstra, 1975)

Horning, 1975). According to Pratt and Zelkowitz:

The difference among programming languages are not quantitative differences in what can be done, by only qualitative differences in how elegantly, easily and effectively things can be done.
(Pratt & Zelkowitz, 2000)

As a concrete example, one can turn to Brian Kernighan's discussion of his preferences between the language PASCAL and C (Kernighan, 1981). Going through the generic features of a programming languages, he comments on the approaches taken by the programming languages on each of these. He professes his preference for the C language, based on their shared inclination for strong typing¹², explicit control flow, cosmetic annoyances and his dislike for an environment in which "*considerable pains must be taken to simulate sensible input*" (Kernighan, 1981). Nonetheless, he acknowledges that PASCAL can nonetheless be a toy language suitable for teaching, thus pointing again the context-dependence of value judgments in programming.

While this example reveals that individual preferences for programming languages can be based on objective criteria when compared to what an ideal language should be able to achieve, Turing-completeness offers an interesting challenge to the Sapir-Whorf hypothesis—if natural languages might only weakly affect the kinds of cognitive structures speakers of those languages can construct, programming languages are claimed to do so to large extents. For instance, Alan Perlis's famous *Epigrams on Programming* mentions that "*A language that doesn't affect the way you think about programming, is not worth knowing.*" (Perlis, 1982). These differences in the ways of doing illustrates how different programming languages are applicable to different domains and different styles of approaching those domains. They do so through different kinds of notations—different aesthetic features—when it comes to realizing the same task.

¹²Something that is, according to him, "*telling the truth about data*" (Kernighan, 1981).

```
puts "hello"
```

Listing 60: A terse example of writing a string to an output in Ruby.

```
import java.io.*;  
  
public class Greeting  
{  
    public static void main(String[] args)  
    {  
        String greeting = "Hey, there";  
        System.out.println(greeting);  
    }  
}
```

Listing 61: A verbose approach to writing a string to an output in Java.

Of the two programs presented in 60 and in 61, the function is exactly the same, but the aesthetic differences are obvious.

The code in 60 is written in Ruby, a language designed by Yukihiro Matsumoto, while the code in 61 is written in Java, designed by James Gosling, both in the mid-1990s. While Ruby is dynamically-typed, interpreted, Java is a statically-typed and compiled language, and both include garbage collection and object-orientation. These two snippets are obviously quite dissimilar at first glance, as the Ruby listing only includes one reserved keyword¹³, puts, while the Java listing involves a lot more lexical scaffolding, including class and function declaration.

From a language design perspective, Robert Sebesta suggests three main features of programming languages in order to be considered good: *abstraction*, *simplicity* and *orthogonality* (Sebesta, 2018). From the two snippets, we now explore some of the most important criteria in programming language design, and how they could underpin the writing of good programs.

¹³Indeed, 60 is also a valid program in Python and Perl, both scripting languages.

Abstraction

Abstraction is the ability of the language to allow for the essential idea of a statement to be expressed without being encumbered by specifics which do not relate directly to the matter at hand, or to any matter at all. Programming languages which facilitate abstraction can lead to more succinct code, and tend to hide complexity (of the machine, and of the language), from the programmer, allowing her to move between different levels of reasoning. For instance, the Java snippet in 61 explicitly states the usage of the `System` object, in order to access its `out` attribute, and then call its `println()` method. While a lot of code here might seem verbose, or superfluous, it is in part due to it being based on an object-oriented paradigm. However, `out` object itself might seem to go particularly contrary to the requirement of programming languages to abstract out unnecessary details: `println()` is a system call whose purpose is to write something on the screen, and therefore already implicitly relates to the output; one shouldn't have to specify it explicitly.

In contrast, Ruby entirely abstracts away the system component of the `print` call, by taking advantage of its status as an interpreted language: the runtime already provides such standard features of the language. Printing, in Java, does not abstract away the machine, while printing, in Ruby, hides it in order to focus on the actual appearance of the message. Another abstraction is that of the language name itself from the import statements. When we write in Java, we (hopefully) know that we write in Java, and therefore probably assume that the default imports come from the Java ecosystem—there shouldn't be any need to explicitly redeclare it. For instance, `System.out.println()` isn't written `java.io.System.out.println()`. Meanwhile, the Ruby listing makes implicit the necessary declaration of `require ".../lib/ruby/3.1.0"`, allowing the programmer to focus, through visual clarity, on the real problem at hand, which the logic of the program being written is supposed to address. In this direction, languages which

provide more abstraction (such as Ruby), or which handle errors in an abstract way (such as Perl) tend to allow for greater readability by focusing on the most import tokens, rather than aggregating system-related and operational visual clutter—also called verbosity.

Related to abstraction is the approach to *typing*, the process of specifying the type of a variable or of a return value (such as integer, string, vector, etc.). A strictly-typed language such as C++ might end up being harder to read because of its verbosity, while a type-free language might be simpler to read and write, but might not provide guarantees of reliability when executed. The tradeoff here is again between being explicit and reliable, and being implicit, subtle, and dangerous (such as JavaScript's very liberal understanding of typing). In some instances, typing can usually be inferred by typographical details: Python's boolean values are capitalized (`True`, `False`), and the difference between string and byte in Go is represented by the use of double-quotes for the former and single-quotes for the latter. In the case above, explicitly having to mention that `greeting` is of type `String` is again redundant, since it is already hinted at by the double-quotes. Ruby does not force programmers to explicitly declare variable types (they can, if they want to), but in this case they let the computers do the heavy lifting of specifying something that is already obvious to the programmer, through a process called dynamic typing.

A particularly note-worthy example of an elegant solution to the trade-off between guarantee of functionality (safety) and readability can be found in some programming languages handling of values returned by functions, such as in the Go listing in 62:

The `_` character which we see on the first line is the choice made by Go's designers to force the user to both acknowledge and ignore the value that is returned by calling the function `getNumber()`. This particular character, acting as an empty line, *represents absence*, not cluttering the layout of the source, while reminding subtly of the *potential* of this particular statement to go wrong and crash the program. Conversely, the functionally equiva-

```

package main

func getNumbers() (int, float64, int) {
    return 1, 2.0, 3
}

func main() {
    first, _, third := getNumbers()
}

```

Listing 62: Go proposes an elegant way of ignoring certain variables, with the use of the underscore token.

```

let getNumbers = () => {
    return [1, 2.0, 3]
}

numbers = getNumbers()
first = numbers[0]
second = numbers[2]

```

Listing 63: JavaScript does not have any built-in syntax to ignore certain variables, resulting in more cumbersome code.

lent code written in JavaScript and shown in 63 does not have this semantic feature (a variable named `_` is still a valid name), and thus requires additional steps to reach the same result.

Abstraction in programming languages is therefore a tradeoff between explicitly highlighting the computer concern (how to operate practically on some data or statement), and hiding anything but the human concern (whether or not that operation is of immediate concern to the problem at hand at all). As such, languages which offer powerful abstractions tend not to stand in the way of the thinking process of the programmer. This particular example of the way in which Go deals with non-needed values is a good example of the designer's explicit stylistic choice.

Orthogonality

Orthogonality is the affordance for a language to offer a small set of simple syntactical constructs which can be recombined in order to achieve greater complexity, while remaining independent from each other¹⁴. A direct consequence of such a feature is the ease with which the programmer can familiarize themselves with the number of constructs in the language, and therefore their ease in using them without resorting to the language's reference, or external program texts under the form of packages, libraries, etc. The orthogonality of a language offers a simple but powerful solution to the complexity of understanding software. Importantly, an orthogonal programming language must make sure that there are no unintended side-effects, such that each program token's action is independent from each other. The functionality of a statement thus comes not just from the individual keywords, but also from their combination.

For instance, the language Lisp treats both data and functions in a similar way, essentially allowing the same construct to be recombined in powerful and elegant ways. To the beginner, however, it might prove confusing to express whole problem domains exclusively with lists. Conversely, the Ruby language makes every data type (themselves abstracted away) an object, therefore making each building block a slightly different version of each other, providing less orthogonality. The silver lining from Ruby's design choice is that it allows for greater creativity in writing code, since everything is an object, which elicits a feeling of familiarity. In turn, this makes the language more habitable, if more uncertain¹⁵.

Orthogonality implies both independence, since all constructs operate distinctly from each other, while remaining related, and cooperation with each other, because their functional restrictions require that be used in

¹⁴An analogy of such affordance is that of the building blocks: for instance, the original LEGO bricks set offers very high orthogonality.

¹⁵The infamous monkey-patching technique of Ruby allows the programmer to even modify standard library functions.

conjunction with one another. This offers a solution to the cognitive burden of programs, in which data can end up being tangled in a non-linear execution, and become ungraspable. This unreadability is triggered, not by verbosity, but because of the uncertainty of, and confusion about, the potential side-effects caused by any statement. Doing one thing, and doing it well, is a generally-accepted measure of quality in software development practices.

Such independence in programming constructs also presents a kind of *symmetry*—a well-accepted aesthetic feature of any artefact—, in that each construct is similar, not in their functionality, but in the fact that their self-contained parts of an orthogonal systems, and therefore share the same quality. This similarity eases the cognitive friction in writing and reading code since an orthogonal language allows the programmer to rely on the fact that everything behaves as stated, without having to keep track of a collection of quirks and arbitrary decisions¹⁶.

Finally, one of the consequences of different amounts of orthogonality is the shift from computer semantic interpretation to human interpretation. Non-orthogonality implies that the compiler (as a procedural representation of the language) has the final say in what can be expressed, reifying seemingly arbitrary design choices, and requiring cognitive effort from the programmer to identify these unwanted interactions, while orthogonal languages leave more leeway to the writer in focusing on the interaction of all programming constructs used, rather than on a subset of those interactions which does not relate to the program's intent.

¹⁶For example, returning an array literal in C is considered illegal syntax, while it is a perfectly common feature of more contemporary programming languages. In this case, the language exhibits an un-orthogonal property since the two constructs (`return` and `int []`) interact with each other in non-independent ways.

Simplicity

Both of these features, abstraction and orthogonality, ultimately relate to simplicity. As Ryan Stansifer puts it:

Simplicity enters in four guises: uniformity (rules are few and simple), generality (a small number of general functions provide as special cases a host of more specialized functions, orthogonality), familiarity (familiar symbols and usages are adopted whenever possible), and brevity (economy of expression is sought). (Stansifer, 1994)

The point of a simple programming language is to not stand in the way of the program being written, or of the problem being addressed. From a language design perspective, simplicity is achieved by letting the programmer do more (or as much) with less, recalling definitions of elegance. This means that the set of syntactical tokens exposed to the writer and reader combine in sufficient ways to enable desired expressiveness, and thus relating back to orthogonality¹⁷.

Moving away from broad language design, and more specific applications, the goal of simplicity is also achieved by having accurate conceptual mappings between computer expression semantics and human semantics (refer to 3.2.3 for a discussion of mappings). If one is to write a program related to an interactive fiction in which sentences are being input and output in C, then the apparently simple data structure `char` of the language reveals itself to be cumbersome and complex when each word and the sentence that the programmer wants to deal with must be present not as sentences nor words, but as series of `char`¹⁸. A simple language does not mean

¹⁷James Rumbaugh describes his conception of simplicity in designing the UML language as such: "If you constantly are faced with four or five alternate ways to model a straightforward situation, it isn't simple" (Biancuzzi & Warden, 2009)

¹⁸Hence the origin of the name of the data type `string`, as a continuous series of `char`, or characters stringed together.

that it is easy¹⁹. By making things simple, but not too simple (Biancuzzi & Warden, 2009), it remains a means to an end, akin to any other tool or instrument²⁰.

A proper combination of orthogonality, abstraction and simplicity results, once more, in elegance. Mobilizing the architectural domain, the language designer Bruce McLennan further presses the point:

There are other reasons that elegance is relevant to a well-engineered programming language. The programming language is something the professional programmer will live with - even live in. It should feel comfortable and safe, like a well-designed home or office; in this way it can contribute to the quality of the activities that take place within it. Would you work better in an oriental garden or a sweatshop? (McLennan, 1997)

Programming languages are thus both tools and environments, and moreover eminently *symbolic*, manipulating and shaping *symbolic* matter. Looking at these languages from a Goodmanian perspective provides a backdrop to examine their communicative and expressive power. From the perspective of the computer, programming languages are unambiguous insofar as any expression or statement will ultimately result in an unambiguous execution by the CPU (if any ambiguity remains, the program does not compile, the ambiguity gets resolved by the compiler, or the program crashes during execution). They are also syntactically disjointed (i.e. clearly distinguishable from one another), but not semantically: two programming tokens can have the same effect under different appearances. The use of formal specifications aims at resolving any possible ambiguity in the syntax of the language in a very clear fashion, but fashionable equiv-

¹⁹Perhaps the simplest language of all being lambda-calculus, is far from an easy construct to grasp, just like the game of Go of which it is said that it is simple to learning, but difficult to master

²⁰For a further parallel on musical instruments, see Rich Hickey's keynote address at RailsConf 2012 (Confreaks & Hickey, 2012)

alence can come back as a desire of the language designer. The semantics of programming languages, as we will see below, also aim at being somewhat disjointed: a variable cannot be of multiple types at the exact same time, even though a function might have multiple signatures in some languages. Finally, programming languages are also differentiated systems since no symbol can refer to two things at the same time.

The tension arises when it comes to the criteria of unambiguity, from a human perspective. The most natural-language-like component of programs, the variable and function names, always have the potential of being ambiguous²¹. We consider this ambiguity both a productive opportunity for creativity, and a hindrance for program reliability. If programming languages are aesthetic symbol systems, then they can allow for expressiveness, first and foremost of computational concepts. It is in the handling of particularly complex concepts that programming languages also differentiate themselves in value. The differences in programming language design and us thus amounts to differences in style. In the words of Niklaus Wirth:

Stylistic arguments may appear to many as irrelevant in a technical environment, because they seem to be merely a matter of taste. I oppose this view, and on the contrary claim that stylistic elements are the most visible parts of a language. They mirror the mind and spirit of the designer very directly, and they are reflected in every program written. (Wirth, 2003)

Idiosyncratic implementations

Software, as an abstract artifact, can be understood at the physical, design and intentional levels(Moor, 1978). With modern programming languages

²¹For instance, does `int numberOffFlowers` refer to the current number of flowers in memory? To the total number of potential of flowers? To a specific kind of number whose denomination is that of a flower?

```
#include <stdio.h>

int main()
{
    int max_count = 5;
    struct int my_list[max_count] = {2046, 2047, 2048, 2049, 2050};

    for (int i = 0; i < max_count; i++)
    {
        printf("%d", my_list[i]);
    }
}
```

Listing 64: Iterating in C involves keeping track of an iterating counter and knowing the maximum value of a list beforehand.

```
my_list = [2046, 2047, 2048, 2049, 2050]
for item in my_list:
    print(item)
```

Listing 65: Iterating in Python is done through a specific syntax which abstracts away the details of the process.

allowing us to safely ignore the hardware level, it is at the interaction of the design (programming) and intentional (human) level that things get complicated; all programming languages can do the same thing, but they all do it in a slightly different way. In order to illustrate the expressivity of programming languages, we highlight three programming concepts which are innate to any modern computing environment, and yet relatively complex to deal with for humans: *iterating*, *referencing* and *threading*.

The first and the most straightforward example is iteration, or the process of counting through the items of a list. Since, ultimately, all program text is organized as continuous series of binary encodings, going through such a list is a fundamental operation in programming. Different implementations of such an operation are shown in 64 for the C language and in 65.

```

int date = 2046;           // `date` refers to the literal value of the
                           // number 2046
int *pointer = &date; // `pointer` refers to the address where the
                     // value of `date` is stored, e.g. 0x5621
*pointer = 1996;          // this accesses the value located at the
                     // memory address held by `pointer` (0x5621) and sets it to 1996
std::cout << date;       // prints the literal value of date, at the
                           // address 0x5621: 1996

```

Listing 66: Pointers involve a non-straightforward way to reason about values.

This comparison shows how a similar function can be performed via different syntaxes. Particularly, we can see how the Python listing implies a more human-readable syntax, getting rid of machine-required punctuation, and thus facilitating the pronunciation out loud. In contrast, the C listing states the parts of the loop in an order that is not intuitive to human comprehension. Read out loud, the C listing would be equivalent to "*For an index named i starting at o, and while i is less than a value named max_count, increase i by one on each iteration*", which focuses more on the index management than on the list itself; while the Python listing would read "*for an item in my list*", much more concise and expressive.

Referencing is a more complex problem than iterating. It is a surface-level consequence of the *use-mention* problem referred to above, the separation between a name and its value, with the two being bound together by the address of the physical location in memory. As somewhat independent entities, it is possible to manipulate them separately, with consequences that are not intuitive to grasp. For instance, when one sees the name of a variable in a program text, is the name referencing the value of the variable, or the location at which this value is stored? Here, we need a mark which allows the programmer to tell the difference.

Notation attempts at remediating those issues by offering symbols to represent these differences, such as in 66, or to hide it completely as in 67.

The characters `*` and `&` are used to signal that one is dealing with a vari-

```

date = 2046
pointer = date
date = 1996
print pointer

# the result is 2046. It is impossible to access the memory location
→ of the variable date

```

Listing 67: Ruby syntax does not allow the programmer to directly manipulate pointers

able of type pointer, and that one is accessing the pointed location of a variable, respectively. Line 2 of the snippet above is an expression called *dereferencing*, a neologism which is perhaps indicative of the lack of existing words for referring²² to that concept. In turns, this hints at a lack of conventional conceptual structures to which we can map such a phenomenon, showing some of the limits of metaphorical tools to think through concepts.

Meanwhile, in 67, we see that the two variables are actually referring to the same data. The design decision here is not to allow the programmer to make the difference between a reference and an actual value, and instead prefer that the programmer constructs programs which, on one side, might be less memory-efficient but are, on the other side, easier to read and write, since variable manipulation only ever occurs in one single way—through reference.

Notation does not exclusively operate at the surface level. Some programming languages signify, by their use of the above characters, that they allow for this direct manipulation, through something called *pointer arithmetic*²³. Indeed, the possibility to add and subtract memory locations independent of the values held in these locations, as well as the ability to do arithmetic operations between an address and its value isn't a process whose meaning comes from a purely experiential or social perspective, but

²²No pun intended.

²³For better or worse, C is very liberal with what can be done with pointers.

rather exists meaningfully for humans only through logical grounding, by understanding the theoretical architecture of the computer. What also transpires from these operations is another dimension of the non-linearity of programming languages, demanding complex mental models to be constructed and updated to anticipate what the program will ultimately result in when executed.

Threading is the ability to do multiple things at the same time, in parallel. The concept itself is simple, to the point that we take it for granted in modern computer applications since the advent of time-sharing systems: we can have a text editor take input and scan that input for typos at the same time, as well as scanning for updates in a linked bibliography file. However, the proper handling of threading when writing and reading software is quite a complex task²⁴.

First, every program is executed as a process. Such a process can then create children subprocesses for which it is responsible. From the hardware standpoint, unpredictability arises from the fact that CPU cores will run different threads of the same process, and yet, as they are under different loads, some processes will get done faster at times and later at other times. The task of the programmer involves figuring out how do the children process communicate information back to the parent process, how do they communicate between each other, and how does the parent process make sure all the children process have exited before exiting itself.

This involves the ability to demultiply the behaviour of routines (whose execution is already non-linear) to keep track of what could be going on at any point in the execution of the program, including use and modification of shared resources, the scheduling of thread start and end, as well as

²⁴As Edward A. Lee puts it: "*Although threads seem to be a small step from sequential computation, in fact, they represent a huge step. They discard the most essential and appealing properties of sequential computation: understandability, predictability, and determinism. Threads, as a model of computation, are wildly non-deterministic, and the job of the programmer becomes one of pruning that nondeterminism.*" (Lee, 2006).

```

package main

import (
    "fmt"
    "math/rand"
    "time"
)

func recall(date int) {
    random_delay := (rand.Int() % 5) + 1
    time.Sleep(time.Second * time.Duration(random_delay))
    fmt.Println(date)
}

func main() {
    recall(2045)
    recall(2046)

    fmt.Println("We're done!")
}

```

Listing 68: Nice way to do threads in Go.

synchronization of race conditions (e.g. if two things happen at the same time, which one happens first, such that the consistence of the global state is preserved?).

For instance, we can look at printing numbers at a random interval. As seen in the non-threaded example in 68, it is somewhat deterministic since we know that 2045 will alway print *before* 2046. In the threaded equivalent in 69, such a result is not guaranteed.

Nonetheless, the threading syntax in 69 allows the programmer to keep their mental modal of a function execution, while the threading syntax in C, shown in 70, creates a lot more cognitive overhead, by declaring specific types, calling a specific function with unknown arguments, and then manually closing the thread afterwards.

Threading shows how the complexity of a deep-structure benefits to be adequately represented in the surface. Once again, aesthetically-satisfying (simple, concise, expressive) notation can help programmers in understanding what is going on in a multi-threaded program, by removing addi-

```

package main

import (
    "fmt"
    "math/rand"
    "time"
)

func recall(date int) {
    random_delay := (rand.Int() % 5) + 1
    time.Sleep(time.Second * time.Duration(random_delay))
    fmt.Println(date)
}

func main() {
    go recall(2046)
    go recall(2047)

    fmt.Println("We're done!")
}

```

Listing 69: Nice way to do threads in Go.

```

#include <iostream>
#include <thread>
#include <pthread.h>
#include <unistd.h>

void recall(int date)
{
    r = (rand() % 5) + 1 sleep(r)
    std::cout << date << '\n';
}

int main()
{
    pthread_t thread1;
    pthread_t thread2;
    pthread_create(&thread1, NULL, recall, 2045);
    pthread_create(&thread2, NULL, recall, 2046);

    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);

    cout << "We're done!";

    return 0;
}

```

Listing 70: Complex way to do threads in C.

tional cognitive overload generated by verbosity.

Here, we see how the abstraction provided by some language constructs in Go result in a simpler and more expressive program text. In this case, the non-essential properties of the thread are abstracted away from programmer concern. The *double-meaning* embedded in the `go` keyword even uses a sensual evocation of moving away (from the main thread) in order to stimulate implicit understanding of what is going on. Meanwhile, the version written in C includes the necessary headers at the top of the file, the explicit type declaration when starting the thread, the call to `pthread_create`, without a clear idea of what the `p` stands for, as well as the final `join()` method call in order to make sure that the parallel thread returns to the main process, and does not create a memory leak in the program once it exits. While both behaviours are the same, the syntax of Go allows for a cleaner and simpler representation.

Programming languages aim at helping programmers solve semantic issues in the problem domain through elegant syntactical means while reducing unnecessary interactions with the underlying technical system. These styles also have a functional component, as we have seen how languages differ in the ways in which they enable the programmer's access to and manipulation of computational actions. Beyond a language designer's perspective, there also exists a social influence on how a source code should be written according to its linguistic community.

5.1.3 Styles and idioms in programming

Concrete use of programming languages operate on a different level of formality: if programming paradigms are top-down strategies specified by the language designers, they are also complemented by the bottom-up tactics of software developers. Such practices crystallize, for instance, in *idiomatic writing*. Idiomaticity refers, in traditional linguistics, to the realized way in which a given language is used, in contrast with its pos-

sible, syntactically-correct and semantically-equivalent, alternatives. For instance, it is idiomatic to say "The hungry dog" in English, but not "The hungered dog" (a correct sentence, whose equivalent is idiomatic in French and German)²⁵. It therefore refers to the way in which a language is a social, experiential construct, relying on intersubjective communication (Voloshinov & Bachtin, 1986). Idiomaticity is therefore not a purely theoretical feature, but first and foremost a social one. This social component in programming languages is therefore related to how one writes a language "properly". In this sense, programming language communities are akin to hobbyists clubs, with their names²⁶ meetups, mascots, conferences and inside-jokes. Writing in a particular language can be due to external requirements, but also to personal preference:

I think a programming language should have a philosophy of helping our thinking, and so Ruby's focus is on productivity and the joy of programming. Other programming languages, for example, focus instead on simplicity, performance, or something like that. Each programming language has a different philosophy and design. If you feel comfortable with Ruby's philosophy, that means Ruby is your language. (Matsumoto, 2019)

So an idiom in a programming language depends on the social interpretation of the formal programming paradigms²⁷. Such an interpretation is also manifested in community-created and community-owned documents.

PEP 20, is one of such documents. Informally titled *The Zen of Python*, it

²⁵In linguistics, we encounter the term "felicitous" to denote utterances that are both grammatically correct and socially accepted; an infelicitous utterance is something a fluent speaker of the language would not say, even if it is grammatically correct.

²⁶Pythonistas for Python, Rubyists for Ruby, Rustaceans for Rust, Gophers for Go, etc.

²⁷This is even more present in contemporary programming languages, since paradigms in these languages are often blended and no language is purely single-paradigmatic; for instance, Ruby is a declarative language with functional properties (Kidd, 2005)

shows how the philosophy of a programming language relates to the practice of programming in it²⁸ (Peters, 1999). Without particular explicit directives, it nonetheless highlights *attitudes* that one should keep in mind and exhibit when writing Python code. Such a document sets the mood and the priorities of the Python community at large (being included in its official guidelines in 2004), and highlights a very perspective on the priorities of theoretical language design. For instance, the first Zen is clearly states the priorities of idiomatic Python:

Beautiful is better than ugly. (Peters, 2004)

This epigram sets the focus on a specific feature of the code, rather than on a specific implementation. With such a broad statements, it also contributes to strengthening the community bonds by creating shared values as folk knowledge. In practice, writing idiomatic code requires not only the awareness of the community standards around such an idiomaticity, but also knowledge of the language construct themselves which differentiate it from different programming languages. In the case of PEP20 quoted about, one can even include it inside the program text with `import this`, showing the tight coupling between abstract statements and concrete code. For instance, in 71, distinct syntactical operators are semantically equivalent but only the second example is considered idiomatic Python, partly because it is *specific* to Python, and because it is more performing than the first example, due to the desire of the developers of Python to encourage idiomaticity; that is, what they consider good Python to be.

Beautiful code, then seems to be a function of knowledge, not just of what the intent of the programmer is, but knowledge of the language itself as a social endeavour. We can see in 72 a more complex example of beautiful, because idiomatic, Python code.

This function calculates the Fibonacci sequence (a classic exercise in

²⁸For equivalent guides in other languages see for instance (Spencer, 1994) or (Cheney, 2019)

```

# idiomatic
for i in range(5):
    print(i)

# generic
for i in range [0, 1, 2, 3, 4, 5]:
    print(i)

```

Listing 71: These two range operators are semantically equivalent in Python, but the first is more idiomatic than the second.

```

@lru_cache(3)
def fib(n):
    return n if n < 2 else fib(n - 1) + fib(n - 2)

```

Listing 72: The decorator is the idiomatic way to calculate the sum of the Fibonacci sequence. (Schmitz, 2015)

computer programming), but makes an idiomatic (and clever) use of decorators in Python. The `@lru_cache(3)` line caches the last 3 results in the least-recently used order, closely mirroring the fact that the Fibonacci sequence only ever needs to compute the terms n , $n-1$ and $n-2$, reducing computational complexity, but at the expense of added complexity for non Pythonistas. Through this, the programmer uses a key, advanced feature of the language in order to make the final program more terse, more precise, and mirroring more faithfully the problem than other implementations, to the detriment of a decrease in readability for non-Pythonistas.

Idiomaticity reflects what the social and aesthetic intent of the language designers and implementers. Notation matters, and designers want to encourage good practices through good notations, assuming that programmers would gravitate towards what is both the most efficient and the best-looking solution.

And it's not hard to "prove" it: If two people write code to solve the same problem and one makes a terrible spaghetti monster

```

if Being.alive
  puts "and well"

if Being.alive?
  puts "and well"

```

Listing 73: Ruby features a lot of syntactic sugar.

in COBOL while the other goes for super-elegant and highly abstracted solution in Haskell, does it really matter to the computer?

As long as the two are compiled to the same machine code, the machine does not care. All the clever constructs used, all the elegance, they are there only to guide our intuition about the code.

(Sustrik, 2021)

Another way to encourage writing good code is through the addition of *syntactic sugar*. Syntactic sugar describes the aesthetic features of the language who are variants of a similar computational feature, and where the only difference between them is their appearance—i.e. visual, semantic shortcuts. The looping examples above are good instances of syntactic sugar, albeit with performance differences. The Ruby language is riddled with syntactic sugar, and highlights how syntactic sugar can "sweeten" the reading process, aiming for more clarity, conciseness, and proximity to natural languages. In Ruby, to access a boolean value on an attribute of an object, one would write it as in any other language. The added syntactic sugar in Ruby comes in the form of the question mark in control flow statements, as shown in 73.

In C, syntactic sugar includes `my_array[i]` to access the `i`th element of the array `my_array`, rather than the more cryptic `*(my_array + i)`. In Python, opening a file could be written as `f = open("notes.md")`, but it also proposes the syntactic sugar of `with open("notes.md") as f:`, which consists in a block which both opens the file, and implicitly closes it at the end of the block.

There are absolutely no functional differences in the statements above,

and the question mark is just here to make the code seem more natural and intuitive to humans. Checking for a boolean (or non-nil value) in an if statement is, in the end, the equivalent of asking a question about that value. Here, Ruby makes that explicit, therefore making it easier to read with the most minimal amount of additional visual noise (i.e. one character).

We have seen how programming languages can be subjected to aesthetic judgment, but those aesthetic criteria are only there to ultimately support the writing of good (i.e. functional and beautiful) code. Such a support exists via design choices (abstraction, orthogonality, simplicity), but also through the practical uses of programming languages, notably in terms of idiomticity and of syntactic sugar, allowing some languages more readability than others. Like all tools, it is their (knowledgeable) use which matters, rather than their design, and it is the problems that they are used to deal with, and the way in which they are dealt with which ultimately informs whether or not a program text in that language will exhibit aesthetic features.

This concept of appropriateness also relates to material honesty. As seen in 4.3.3, the fact that a programmer tends to identify their practice with craft implies that they work with tools and materials. Programming languages being their tools, and computation the material, one can extend to the concept of material honesty to the source code (Sennett, 2009). In this case, working with, and in respect of, the material and tools at hand is a display of excellence in the community of practitioners, and results in an artefact which is in harmony and is well-adapted to the technical environment which allowed it to be. Source code written in accordance with the principles and the affordances of its programming language is therefore more prone to receive a positive aesthetic judgment. Furthermore, idiomatic writing is accompanied by a language-independent, but group-dependent feature: that of programming style.

Fundamentally, the problem of style might be that "*the practical ex-*

istence of humanity is absorbed in the struggle between individuality and generality" (Simmel, 1991). Simmel's investigation of the topic originally focuses on the dichotomy between works of fine art and mass-produced works of applied arts. Indeed, Simmel draws a distinction between the former, as idiosyncratic objects displaying the subjectivity of its maker, and the latter, as industrially produced and replicated, in which the copy cannot be told apart from the original. The work of fine art, according to him, is *a world unto itself, is its own end, symbolizing by its very frame that it refuses any participation in the movements of a practical life beyond itself*, while the work of applied arts only exists beyond this individuality, first and foremost as a practical object.

As these two kinds of work exist at the opposite extremes of a single continuum, we can insert a third approach: that of the crafted object. It exists in-between, as a repeated display of its maker's subjectivity, destined for active use rather than passive contemplation (Sennett, 2009). So while style can be seen as a general principle which either mixes with, replaces or displaces individuality, style in programming doesn't stand neatly at either extreme. The work of Gilles-Gaston Granger, and his focus on style as a structuring practice can help to better apprehend style as a relationship between individual taste and structural organization (Granger, 1988). Granger posits style in scientific endeavours, which is a component of programming practice, as a mode of knowing at the scale of the group. Abiding by a particular style, the writer and reader can implicitly agree on the fundamental values underpinning a given text, and thus facilitate expectations in further readings of a given program text.

Concretely, programming style exist as dynamic documents, with both social and technical components. On the social side, they are only useful if unconditionally adopted by all members working on a particular code-base, since "*all code in any code-base should look like a single person typed it, no matter how many people contributed.*" (Waldron, 2020); personal style is usually frowned upon by software developers as an indicator of individual

preferences over group coordination²⁹.

In the strict sense, guidelines are therefore reference documents which should provide an answer to the question of what is the preferred way of writing a particular statement (e.g. var vs. let, or camelCase vs. snake_case). Beyond aesthetic preferences aimed at optimizing the clarity of a given source code, style guides also include a technical component which aims at reducing programming errors by catching erroneous patterns in a given codebase (e.g. variable declaration before initialization, loose reference to the function-calling context).

Programming style also exhibits the particular property that it is not just enforced by convention, but also by computational procedure: linters and formatters are particular software whose main function is to formally rearrange the appearance of lines of code according to some preset rules. This constitutes an additional socio-technical context which further enmeshes human writing and machine writing (Depaz, 2022). Essentially, this means that source code will be judged not just on how it functions technically, but also how it exists stylistically—that is, within a social contract which can be implemented through technical, automated means.

In conclusion, programming languages, as a symbol systems subject to aesthetic judgment, are an important factor in allowing for aesthetic properties to emerge during the process of writing program texts. They present affordances for the abstraction and combination of otherwise-complex programming concepts, for the development of familiarity through their idiomatic uses and for ease of readability—to the point that it might be

²⁹Angus Croll wrote a satirical book, *What if Hemingway Wrote JavaScript*, about personal style in programming, in which he copies the style of fiction authors into different programming languages (Croll, 2014). This shows that, while personal style and expression is very much possible in programming languages, it is also somewhat ludicrous

come transparent to experienced readers. Yet, one must keep in mind that there is a difference between considering a programming language good or beautiful *in itself*, considering the quality of the programs written related to the programming language they are written in, in more general aesthetic features. In the next section, we look at some of those aesthetic features which can be transposed across languages.

5.2 Cognitive aesthetics in program texts

In this section, we show how the aesthetics of source code can be understood through the dual lenses of spatial navigation and semantic compression. We start by highlighting how previous scholars have engaged with the semantic ambiguities that source code presents, including linguistic, poetic and functional perspectives.

We complement this approach by suggesting that semantic compression is tied to the spatial navigation of a program text—i.e. its non-linear active reading patterns. To do so, we will see how we can consider source code aesthetics along a logic of levels; working from structure across syntax and towards vocabulary, these different levels have different connotation in terms of levels of abstraction. Positively valued aesthetic manifestations at each of these levels facilitate reasoning about more or less abstract parts³⁰ of the program text. Aesthetic manifestations of source code provide different levels of granularity when it comes to describing the what, how and why of a program.

Furthermore, we show how this is complemented by semantic compression, understood as the ability to reference concepts from multiple domains (hardware, software, problem) in order to both minimize the amount of cognitive effort necessary to grasp all implications denoted by a given token. As different practices of different programmers might pri-

³⁰The most abstract level of a program text is considered to be its data modelling of the problem domain, while the least abstract is considered to be the hardware operations.

oritize different aspects to source code aesthetics, we provide an overview of how values such as abstraction, transparency, openness, function and emotion are best seen in certain kinds of program texts, but are not exclusive to them.

5.2.1 Between humans and machines

The ambivalence of source code has also been explored in the literature through different names. As we will see, all of these argue for the intertwining of human interpretation and machine execution. This ambivalence is first taken up by Mateas and Montfort in their study of weird programming languages (Mateas & Montfort, 2005); in it, they highlight an aesthetics of code that goes beyond the mainstream "literate programming" (see 2.3.1). Rather than making clear and elegant, they inquire about the aesthetic effects of obfuscation in esoteric languages, one which departs from a requirement of source code to be understandable to both humans and computers, and ultimately argue that esoteric languages do so by playing with more traditional understanding of double-coding as "open to multiple interpretations"³¹.

Also focusing on the more fringe and creative uses of code, Camille Paloqué-Bergès presents the related concept of *double-meaning* in her work on networked texts and code poetics (Paloqué-Bergès, 2009). She defines it as the affordance provided by the English-like syntax of keywords reserved for programming to act as natural-language signifiers. As we've seen in *Black Perl* (22), the Perl functions can indeed be interpreted as regular words when the source is read as a human text. As she continues her analysis of *codeworks*, a body of literature centered around a créole lan-

³¹"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 coding 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).

guage halfway between humanspeak and computerspeak³², it can be extended into the aesthetically productive overlap of syntactic realms, however leaving aside any functional or productive aspect of source code.

Previous research by Philippe Bootz has also highlighted the concept of the *double-text* in the context of computer poetry, a text which exists both in its prototypal, virtual, imagined form, under its source manifestation, and which exists as an executed, instantiated, realized one (Bootz, 2005). However, he asserts that, in its virtual form, "a work has no reality", specifically because it is not realized. Here, we encounter the dependence of the source on its realized output, indeed a defining feature of the generative aesthetics of computer poetry. A work of code poetry can very much exist as a prototypal form, with its output providing only additional meaning, further qualifying the themes laid out in source beforehand. From this perspective, the output of a code poem would have a drastically diminished semantic richness if the source is only read, or only executed. For this double-meaning to take place, we can say that the situation is inverted: the output becomes the virtual, imagined text, while the source is the concrete instantiation of the poem.

The role of execution is even more embedded in Geoff Cox and Alex McLean's take of *double-coding* (Cox & McLean, 2013). According to them, double-coding "*exemplifies the material aspects of code both on a functional and an expressive level*" (p.9). Cox and McLean's work, in a thorough exploration of source code as an expressive medium, focus on the political features of speaking through code, as a subversive praxis. They work on the broad social implications of written and spoken code, rather than exclusively on the specific features of what makes source code expressive in the first place, with a particular attention to the practice of live coding. Double-coding nonetheless helps us identify the unique structural features of programming languages which support this expressivity, such as reserved keywords, data types and control flow. As we show below, no-

³²See in particular the work of Alan Sondheim and mezangelle

tably through the use of data types such as symbols and arrays in source code poetry, programming languages and their syntax hold within them a specific kind of semantics which enable, for those who are familiar with them and understand them, expressive power, once the computer semantics are understood both in their literal sense, and in their metaphorical sense. The succinct and relevant use of these linguistic features can thicken the meaning of a program text and, in the case of code poetry, bringing into the realm of the thinkable ways to approach metaphysical topics.

Finally, the tight coupling of the source code and the executed result brings up Ian Bogost's concept of *procedural rhetoric* (Bogost, 2008). Bogost presents procedures as a novel means of persuasion, along verbal and visual rhetorics. Working within the realm of videogames, he outlines that the design and execution of processes afford particular stances which, in turn, influence a specific worldview, and therefore argue for the validity of such worldview. Further work has shown that source code examination can already represent these procedures, and hence construct a potential dynamic world from the source (Tirrell, 2012; Brock, 2019). If procedures are expressive, if they can map to particular versions of a world which the player/reader experiences, then it can be said that their textual description can also already be persuasive, and elicit both rational and emotional reactions due to their depiction of higher-order concepts (e.g. consumption, urbanism, identity, morality). As its prototypal version, source code acts as the pre-requisite for such a rhetoric, and part of its expressive power lies in the procedures it deploys (whether from value assignment, execution jumps or from its overall paradigms). Manifested at the surface level through code, these procedures however run deeper into the conceptual structure of the program text, and such conceptual structures can nonetheless be echoed in the lived experiences of the reader.

The ambivalence between human meaning and machine meaning is thus at the core of source code aesthetics. We develop on this work in our analysis below by showing some of the configurations of source

code which can elicit an aesthetic experience during reading and writing. Through the investigation of levels of abstraction, spatial navigation, metaphorical expression and functional correspondence, we offer concrete examples of the different ways semantic layerings can be manifested in program texts.

5.2.2 Matters of scale

The discourses of programmers in our corpus do not contain unidimensional criteria, but rather criteria which can be applied at multiple levels of reading. Some tend to relate more to the over-arching design of the code examined while others focus on the specific formal features exhibited by a given token or successions of tokens in a source code snippet. To address this variability of focus, we borrow from John Cayley's distinction between structures, syntaxes and vocabularies (Cayley, 2012). Cayley's framework will allow us to take into account an essential aspect of source code: that of scales at which aesthetic judgment operates. From the psychological accounts of understanding source code in 3.2.3 to the uses of space in domain-specific aesthetics in 4.2.3 or 2.3.3, one of the specificities of source is the multiple dimensions of its deep structure hidden behind the two-dimensional layout of a text file, and the need for programmers to navigate such space.

Structure is defined by the relative location of a particular statement within the broader context of the program text, as well as the groupings of particular statements in relation to each other and in relation to the other groups of statements within the program-text, whether it is across the same file, series of files, or a sprawling network of folders and files. This also includes questions of formatting, indenting and linting as purely pattern-based formal arrangements, as seen in 3.3.2, since these affect the whole of the program-text.

Syntax concerns the local arrangement of tokens within a statement

or a block statement, including control flow, iterators statements, function declarations, etc., which can be referred to as the "building blocks" of a program-text. Syntax also includes language-specific choices—idioms—and generally the type of statements needed to best represent the task required (e.g. using an array or a struct as a particular data structure for representing entities from the problem domain).

Finally, the vocabulary refers to the user-defined elements of the source code, in the form of variable, function, classe and interface names. Source code vocabulary is constituted of both reserved keywords (which the computer "understands" by being explicitly mentioned by the language designers) and user-defined keywords, the single words which the writes defines themselves and which are not known to a reader who would already know the language's keywords. Unlike the two precedent categories, this is therefore the only one where the writer can come up with new tokens, and is the closest to metaphors in traditional literature.

Structure

At the highest-level, the structure of a program-text can be examined at the surface-level and at the deep-level. The criteria for beauty in surface-structure is layout, as the spatial organization of statements, through the use of line breaks and indentations. While serving additional ends towards understanding, proper layout (whether according to stylistic conventions, or deliberately positioning themselves against these conventions) seems to be the first requirement for beautiful code. In terms of aiding understanding, blank space creates semantic groupings which enable the reader to grasp, at a glance, what are the decisive moments (Sennett, 2009) in the code's execution, and presented by some as akin to paragraphs in litterature (Matsumoto, 2007). Such groupings also fit Détienne's identification of *beacons* (Detienne, 2001) as visual markers that indicate important actions in the program text, whether these actions can be described in a sin-

gle line, in a block or, more rarely, in a series of blocks. Any cursory reading of source code always first and foremost judges layout.

This aid to understanding is further highlighted by a deep-structure criteria of *conceptual distancing*; statements that have to do with each other are either located close to each other or, in the case of more complex program texts, are separated in coherent units (such as in folders) and connected by single syntactic expressions, such as `import` or `use` statements. Such statements establish an intratextual dimension insofar as it acts as an alias for a larger piece of code, making the most of practices of abstraction and of non-linear readings. As such, visual appearance at the level of the file can reflect the conceptual structure of the code. At the level of the folder(s), that is, at the level of a collection of files located at different levels of nestedness, one can also highlight stylistic agreement or disagreements. In conventional architectures such as the Model-View-Controller, or the use of `lib`, `bin` or `data` folders also act as aesthetic makers establishing the mental space of the programmer ahead of the reading of the actual source code³³.

Another instance of intratextual interfacing is the *limitation of function arguments*, according to which arguments given to a function should be either few, or grouped under another name. Going back to the structural criterion above of limiting input/output and keeping groups of statements conceptually independent, function arguments solve this requirement at the level of vocabulary, demonstrating in passing the relative porosity of those categories. Indeed, the naming of variables also reveals the pick of *adequate data-structures*, echoing those who claim that the data on which the code operates can never be ignored, and that beautiful code is code which takes into account that data and communicates it, and its mutations, in the clearest, most intelligible, possible way. Such an echo relates to our discussion on the issues of problem-domain modelling, analyzed in 3.2.2;

³³An extreme example of this can be seen in the structure of Ruby on Rails applications, famous for its reliance of convention of folder and file naming.

ultimately, the structure, syntax and vocabulary of a program text has a necessary involvement with the problem domain.

Within the file, structural aesthetic criteria are vague enough to be open to interpretation by practitioners and is therefore unable to act as a strict normative criteria, but are nonetheless a solid heuristic of the quality of the software. For instance, a program text should follow the stepdown rule of function declaration rather than the alphabetical rule when writing in a language which doesn't enforce it. As for variable declarations, global variables should all be declared at the beginning of the highest scope to which they belong (e.g. at the beginning of the file), rather than at the closest location of their next use. Program-texts therefore tend to be more aesthetically pleasing when semantic groupings are respected by the human writer (such as global variable declaration) and syntactic groupings are respected by the machine writer (through linting and formatting).

This uncovers the related criteria of *local coherence*: what is next to each other should relate to each other. Local coherence reveals what Goodman calls semantic density, in which tokens grouped together obtain a greater denotative power, while remaining open to further modification by inserting modifications within this grouping. Local coherence operates in balance with the undesirable but unavoidable entanglement of code, as proponents of local coherence in source code imply that a beautiful piece of code should not have to rely on input and output (i.e. not be entangled) and therefore be entirely autotelic. Such an assumption runs contrary to the reality of software development as a practice, and as an object embedded in the world, and thus not "usable" by software developers. This balancing issue can be resolved by writing a source code whose code blocks are structured in such a way that they are related to, but not dependent on, each other³⁴.

³⁴This aesthetic standard can be traced back to what is called the UNIX philosophy: a related set of tools which only do one thing, do it well, and can be combined into more complex structures.

As we travel down from structure to syntax, we can point to a correlate to conceptual distancing in the form of *conceptual symmetry*, according to which that groups of statement which do the same thing should look the same. It then becomes possible to catch a glimpse of patterns, in which readers get a grasp of what such pattern does. Conceptual distancing can be further improved by *conceptual uniqueness*, which demands that all the statements that are grouped together only refer to one single action: complex enough to be useful, and simple enough to be graspable. Aesthetically-pleasing code is thus the code that "does the job" while using the least amount of different ideas, which implies a linear relationship between the number of lines of code and the amount of conceptual information to be understood³⁵, and a relation to elegance.

Finally, it should be noted that this aesthetic criteria of structure is most relevant for a particular class of program texts, texts written by software engineers. In the case of hackers, poets or scientists³⁶, their program texts are limited in numbers of lines when compared to code bases of, for instance, large open-source projects. And yet, as we will see below, in their case, aesthetic code is still code which manages to pack the maximum number of ideas in a minimal amount of lines of code, both in obfuscation practices, one-liners, poetic depictions and in demonstrations of algorithmic ideas.

Syntax

Syntax, as the mid-level group of criteria, deals most explicitly with two important components of the implementation: the algorithm and the programming language. Beautiful syntax seems to denote a conceptual understanding of the computational entities and of the tools at hand to solve

³⁵In software engineering, this is referred to as the DRY principle—Do not Repeat Yourself(Martin, 2008)

³⁶In the case where scientists write toy or educational programs, without relying on software engineering practices, as seen in 2.1.3.

a particular problem, and implies an expertise (i.e. both a knowing-what and a knowing-how).

Here, we consider both algorithms and languages as tools since they are part of the implementation implementation process: a process which turns an idea into form and ensures that this form is functional, and can thus subsequently be examined for aesthetic purposes. While algorithms exist independently from languages, their aesthetic value in the context of this research cannot be separated from the way they are written, itself affected by the language they are written in. Indeed, most algorithms are expressed first as pseudo-code and then implemented in the language that is most suited to a variety of factors (e.g. speed, familiarity of the author, suitability of the syntax, nature of the intended audience); this seems to be a contemporary version of the 1950s, when computer scientists would devise those algorithms through pencil and paper, and then leave their implementation at the hands of entirely different individuals—women computers (Chun, 2005).

Beautiful syntax in code responds to this limitation. It aims at resolving the tension between clarity and complexity, with the intent to minimize the number of lines of code, while maximizing both the conceptual implications and the specific affordances for modification. Since algorithms must be implemented in a certain context, with a certain language, it is the task of the writer to best do so with respect to the language that they are currently working in. In this case, knowledge of the language from both writers and readers makes *idiomatic syntax* a beautiful syntax (see 5.1.3 above). This involves knowing the possibilities that a given language offers and, in the spirit of the craftsmanship ethos noted previously, working with the language rather than against it. These sets of aesthetic criteria thus become dependent on the syntactical context of the language, itself dependent on its suitability for the problem at hand, and can only be established with regards to each languages. Specifically, this involves knowing which keywords should traditionally not be used, such as `unless` in perl, or

* in C, knowing when to use decorators in python, or the spread ... operator in ECMAScript, etc. A common feature shared by these keywords is their tendency to cause more cognitive friction than ease of comprehension.

Here, syntax also engages with the ideal of conciseness: a writer can only be concise if they know how the language enables them to be concise. Knowing the algorithms implemented and the problem domain addressed also influence the overall experience of the program text, as the goal is to optimize these three components. The extent to which a syntax is idiomatic and the extent to which the problem domain is accurately represented³⁷, are therefore good indicators of the aesthetic value of a program-text. Conversely, quality syntax is also syntax which refrains from being too idiomatic for the purposes at hand, in software engineering; this is referred to as "clever code" and is generally frowned upon. In the case of hacking code or poetry, cognitive friction is, on the opposite, seen as a positive aesthetic experience. By doing the most with the least, complex hacker code and code poetry enable understandings of polar opposites: hacker syntax displays insight into the highly technical hardware or machine-linguistic environment, while poet syntax offers access to broad human concepts (e.g. of the self, of religion, or history) through a minimal number of lines of code.

A programmer who finds that she can best communicate her ideas according to Java will find Java beautiful. A developer who finds that she can best communicate her ideas while writing in Go will find Go beautiful, and so on³⁸. Ultimately, a syntactical criteria which acts as a response to these discussions is *consistency*. While there might be specific, personal, preferences as to why one would want to be writing code one way or an-

³⁷An accurate representation of the problem domain is a representation which enables a direct cognitive mapping between machine-syntactic tokens and human mental models of the entities of the problem domain being processed

³⁸This state of affairs seems to be part of the reason as to why online platforms are full of endless discussions around the question of which language is better.

other (e.g. calling functions on objects rather than calling functions from objects in order to prevent output arguments), this minor increase in aesthetic value through subjective satisfaction—through display of individual skill and personal knowledge—does not compensate for the possible increase in cognitive noise in a collaborative environment. If those different ways of writing are used alternatively in an arbitrary manner, this requires unnecessary mental gymnastics from the reader. In this context, consistency prevails over efficacy, and confirms at the fact that aesthetics in source code in this context is a game of tradeoffs. Again, hacker and poet aesthetics stand at the opposite: the highly localized and personal function of the program text implies a tolerance for idiosyncracy, since personal knowledge and preference are part of the aesthetic value of the program text.

Beyond the state of syntactic consistency and idiomatic writing, another aesthetic criteria is *linguistic reference*, meaning bringing practices from one language into another. Being able to implicitly reference another language in a program-text³⁹, a code-switching of sorts, can both communicate a deep understanding of not just a language, but an ecosystem of languages while satisfying the purpose of maintaining clarity, at the expense of, again, assuming a certain skill level in the reader. This communicates a feeling of higher-understanding, akin to perceiving all programming languages as ultimately just "tools for the job" and whose purpose is always to get a concept across minds as fully and clearly as possible. However, a misguided intention of switching between two languages, or a mis-handled implementation can push a program-text further down the gradient of ugliness. The concept communicated would in such a case be obscured by the conflicting idioms, reveal of lack of mastery of the unique aspects of the working language(s), and therefore fail to fulfill the aesthetic

³⁹e.g. "this is how we do it now that we have C++, but the current code is written in C, so one can bring in ideas and syntax that are native to C++" or "since Ruby can qualify as a Lisp-like language, one can write lambda functions in an otherwise object-oriented language"

criterion of being true to ones material.

Moving down to the level of vocabulary, a final syntactic criterion with high aesthetic value is the preference of *natural language reading flow*. For instance, of the two alternatives in Ruby: `if people.include? person` vs. `if person.in? people`, the second one is to be considered more beautiful than the first one, since it adapts to the reader's habit of reading human languages. However, the essential succinctness and clarity of source code is not to be sacrificed for the sake of human-like reading qualities, such as when writers tend to be overly explicit in their writing. Indeed, a definite criteria for ugliness in program-text is verbosity, or useless addition of statements without equivalent addition of functionality or clarity. This is, once again, an example of source code aesthetics being a balance between machine idioms over human idioms—here, the resolution of this balance is the point at which machine idioms are presented as human-readable.

Vocabulary

Vocabulary, as the only component in this framework which directly involves words that can be chosen by the writers themselves, is often the most looked at in the literature regarding beautiful code among software developers, as it is the closest to human aesthetics, and since their understanding does not require existing knowledge of programming, but also the easiest to assess their functional impact (Oliveira et al., 2022). Aesthetics here exist at the level of the name and affects most directly the readership of a program text.

Of the two big problems of programming, the most frequent one is naming⁴⁰. One reason as to why that is might be that naming is an inherently social activity, because a name is an utterance which only makes sense when done in the expectation of someone else's comprehension of that

⁴⁰The full statement, that : "There are only two hard things in Computer Science: cache invalidation and naming things." is a piece of folk knowledge generally attributed to Phil Karlton (Fowler, 2009)

name (Voloshinov & Bachtin, 1986). This is supported by the fact that the process of creating a variable or function name on one's own is often more time-consuming when done alone, as opposed to discussing it with others. Naming, furthermore, aims not just at describing, but at capturing the essence of an object, or of a concept. This is a process that is already familiar in literary studies, particularly in the role of poetry in naming the elusive. Here, we remember how Vilém Flusser sees poetry as the bringing-forth that which is conceivable but not yet speakable through its essence in order to make it speakable through prose, using the process of naming through poetry in order to allow for its use and function in prose (see 2.3.1 for a mention of Flusser's conception of prose and poetry). In this light, good, efficient and beautiful names in code are those who can communicate the essence of the concept that is being operated upon in the program text, implying both what they are and how they are used, while omitting extraneous details.

On a purely sensory level, surface-level aesthetic criteria related to naming are that of *character length* and *pronounceability*. Visually, character length can indicate the relative importance of a named concept within the greater structure of the program-text. Variables with longer names are variables that are more important, demand more cognitive attention, offer greater intelligibility in comparison with shorter variable names, which only need to be "stored in memory" of the reader for a smaller amount of time. This length also signifies at which level of abstraction is the variable operating: longer names denote more global variables that denote the program text's structure, while shorter variable names indicate that we are currently working at a lower level of abstraction. Variables and functions with longer names thus exist in a broader scope than their counterparts with shorter names, re-introducing a component of scale within our existing framework of scale.

Pronounceability, meanwhile, takes into account the basic human action of "speaking into one's head", as an internal dialogue, and there-

fore participates in the requirement for communicability of source code amongst human readers. For instance, the difference between `mnPtCld` and `meanPointCloud` both refer to the same entity, and the second provides an easier cognitive access to it at the very minimal expense of a few characters.

Equally visual, but aesthetically pleasing for both typographical and cognitive reasons, is the *casing* of names. Dealing with the constraint that variable names cannot have whitespace characters as part of them, casing has resulted into the establishment of conventions which pre-exists the precise understanding of what a word denotes, by first bringing that word into a category (all-caps denotes a constant, camelCasing denotes a multi-word variable and first-capitalized words indicate classes or interfaces). By using multiple cues (here, typographical, then semantical), casing helps with understandability and, in this specific instance, there seems to be quantitative evidence for CamelCasing to facilitate the scanning of a program text (Binkley et al., 2009). Again, casing, by its existence first as a convention, implies that it exists within a social community of writers and readers, and acknowledges the mutual belonging of both writer and reader to such a community, and turns the program-text from a readerly text further into a writerly one (Barthes, 1984).

Following these visual, auditory and typographical criteria, an aesthetically-pleasing vocabulary includes a strict naming of *functions as verbs and variables as nouns*. Continuing this correspondence between machine language and human language, there is here a clear mapping between syntax and semantics: functions do things and variables are things. If written the other way around, while this would respect the criteria for consistency, functions as nouns and variables as verbs hint at what it is not, are counter-intuitive and ultimately confusing—confusion which brings ugliness.

The noun given to a variable should be a hint towards the concept addressed, and ideally address what it is, how it is used, and why it is present,

things that cannot be deduced from the environment of the program text⁴¹. Each of these three aims are not necessarily easily achieved at the same time, but finding one word which, through multiple means, points to the same end, is an aesthetic goal of source code writers and another testimony of elegant writing. A beautiful name is a name which differentiates between value (obvious, decontextualized, and therefore unhelpful) and intention, informing the reader not just about the current use, but also about future possible use, in code that is written or yet to be written. This is particularly salient in the general distaste of the use of magic numbers, as are called pure values, which do not have a semantic label applied to them. We see here a paradox between direct conceptual relationship between a name and what it denotes, and the multiple meanings that it embodies (its description, its desired immediate behaviour, and its purpose).

Indeed, in the community of software developers, variable names should then have a direct mapping with the object or concept they denote. This is not the case in other communities, whether those that rely on obfuscation, in which confusion becomes beautiful, or in poetic code, in which *double-meaning* brings an additional, different understanding which ultimately enriches the complexity of the reading, by moving it away from strict functionality.

The contextual nature of source code aesthetics proposes a slightly adjacent standard for source code poetry, in which the layering of meanings is a positive aesthetic trait in the community of code poets. These ambivalent semantics allow writers to offer metaphors, and provide an entry point to the metaphorical tendencies of source code. This aesthetic criteria of double-meaning comes from poetry in human languages, in which layered meanings are aesthetically pleasing, because they point to the unutterable, and as such, perhaps, the sublime(Aquilina, 2015).

⁴¹For instance, in statically typed languages, one does not include the type of the variable in its name, since it is enforced by the compiler

Comments

Before moving on to another aspect in which the aesthetics of source code involved with spatialization of meaning mechanically-realized, we touch on a specific case of machine languages. Comments in code do not seem to fall clearly in any of the three categories above. By definition ignored by the compiler or interpreter, comments can be erroneous statements which will persist in an otherwise functional codebase, and are therefore not entirely trusted by experienced, professional software practitioners. In this configuration, comments seem to exist as a compensation for a lack of functional aesthetic exchange.

By functional aesthetic exchange we mean an exchange in which a skilled writer is able to be understood by a skilled reader with regards to what is being done and how, only through executable source code. If any of these conditions fail (the writer isn't skilled enough and relies on comments to explain what is going on and how it is happening, or the reader isn't skilled enough to understand it without comments), then comments are here to remedy to that failure, and therefore are a symptom of non-beautiful code, specifically because it relies on extraneous devices, that the computer does without. Nonetheless, they can act as a locus for social expression, or human creativity—⁷⁴ is an example of such a display of ingenuity and helpful mental scaffolding for understanding the code.

The situation in which comments seem to be tolerated is when they provide contextual information, therefore (re-)anchoring the code in a broader world. For instance, this is achieved by offering an indication as to why a given action is being taken at a particular moment of the code, called contractual comments, again pointing at the social existence of source code. This particular use of comments seems to bypass the aesthetic criteria of source code being self-explanatory. However, it also integrates the criteria of being writable, a piece of code which, by its appearance, invites the reader to contribute to it, to modify it. As such, in an educational set-

Listing 74: An example of useful comments complementing the source code in Mozilla's layout engine, literally drawing out the graphical task executed by the code.

ting (from a classroom to an open-source project), comments are welcome, but rarely quoted as criteria for beautiful code in other communities.

In conclusion, we have seen that aesthetic standards for source code can be laid out along a logic of scale, from a macro-level of structure all the way to a micro-level of vocabulary, through an appropriate use of syntax. Throughout, the aesthetic principles of consistency, elegance and idiomacity are recurring concepts against which a value judgment can be given. Such aesthetic manifestations enable the traversal of the program text, from the micro- to the macro-, from file to file, class to class, or layer to layer. However, this approach of a linear scale also hints at another dimension: that of the interface between human concepts and machine concepts, as source code aesthetics enhance the communication of separate semantic layers.

5.2.3 Semantic layers

The specificity of source code is that it acts as a techno-linguistic interface between two meaning-makers: the human and the machine. While the machine has a very precise, operational definition of meaning (see 5.1.1 above), programmers tend to mobilise different modalities in order to make sense of the system they are presented with through this textual interface (see 3.2.3). Among those modalities are the resort to literary techniques (in the form of metaphors), to architecture (in the form of pattern-based structural organization), to mathematics (in the form of symbolic elegance) and craft (in the form of material adequacy and reliability).

As a formal manifestation involving, in the context of a crafted object, a producer and a receiver, aesthetics contribute to the establishment of mental spaces. In domains such as mathematics or literature, mental spaces can represent theorems or emotions; within source code, however, they acquire a more functional dimension. As such, they also communicate *states* and *processes*.

Building on our discussion of understanding software in 3.2, we now highlight concrete instances of complex computational objects interfaced through source code. We take three examples to highlight the multiple manifestations of semantic layers at play in source code, operating in different socio-technical contexts, yet all sharing the same properties of using structure, syntax and vocabulary in order to communicate implicitly a relatively complex idea. We can consider a semantic layer to be an abstraction over relatively disorganized data in order to render it relatively organized, by providing specific reference points which contribute to the establishment of a mental space, based on the computational space of the program. In the words of Peter Neumann:

A challenge in computer system design is that the representation of the functionality at any particular layer of abstraction should exhibit just those characteristics that are essential at that layer, without the clutter of notational obfuscation and unnecessary appearance of underlying complexity. (Neumann, 1990)

First, we look at the abstract data type called a *semaphore*, and how it operates as the interface between the computational reality of concurrency, and the human associations of traffic, and resorting primarily from the scientific domain. Second, we turn to a fragment of open-source software which uses abstraction and configuration in order to signify modification, in the context of collaborative, read-write texts characteristic of software development. Finally, we discuss a code poem, as an illustration for the role of programming languages and creative metaphors to support a different kind of functional communication, in Goodman's sense of *working*. Each of these examples have been chosen to reflect the different practices of programming, and should thus be considered in a complementary manner, rather than independently. While each was chosen for how well it illustrates their respective, concept, abstraction, transparency and execution are all parameters that come into play for each practices of program-

ming..

Programming can be seen as an act of encryption and decryption across layers, in which human meaning is encrypted in machine language for computational execution, with the possibility for such meaning of being decrypted later on for study and modification (Ledgard, 2011). We argue that this encryption process involves different aesthetic modalities which act as a heuristic for writing functionally good code and provide keys for decrypting the intent and processes represented in source code, moving across human and machine layers. Each in their own way, these modalities represent different semantic layers which bridge machine-meaning and human-meaning.

Abstraction and metaphors

An early and recurring problem in computer science is that of concurrency. Concurrency, or the overlapping execution of multiple interacting computational tasks, emerged along with the development of time-sharing and multi-core hardware. From the 1964 development of MULTICS, a time-sharing operating system that multiple users could use at the same time, to the popularization of multi-processor architecture in the early 1980s, computers moved from executing one task for one user at a time, to multiple tasks for multiple users. The issue that arises then is that of shared memory: how can one design a program in which two parallel operations can access and modify a shared resource, while at the same time guaranteeing the integrity of the data?

While this problem arose from hardware development, whether synchronization is of a universal nature—that is, whether there exists a solution which can be applied to all practical synchronization problems—has been an ongoing research investigation (Leppäjärvi, 2008). This tension between hardware innovation and fundamental computing problems illustrates the multiplicity of layers at play, from matter to ideal, in pro-

gramming practice.

Specifically for concurrency, it turns out to be quite difficult for human programmer to model the different actions taking place in parallel, with overlapping consequences on common data. Just like programming languages can nudge their users into safety⁴², there are technical systems which can be designed in order to help humans both think through and implement mechanically this thinking. One such system is Edsger Dijkstra's *semaphores*.

Writing in 1965, Dijkstra describes a data type which prevents such issues of simultaneous access to critical data by two threads of a same process. Such data type possesses two behaviours: `post` and `wait`. When a thread is about to access a critical part of data, it calls `wait`, and when it is done, it calls `post`. If another thread also calls `wait` before accessing the critical part, `wait` checks its internal value to see if another thread is currently processing that data. If it is, then it puts the requesting thread to sleep, and waits until `post` is called to wake it up (Dijkstra, 1965). A textbook implementation of such data structure is described in 75.

For instance, say two separates users (P and J) want watch an online video at the same time, and the website wants to keep track of the number of views. If `viewCount` is the variable of the number of views, and is equal to 2046 before the visit. Parallel execution means that, if P and J view the video at the same time, they might both separately increase the `viewCount` from 2046 to 2047, rather than one waiting for the other to complete the increase, and end up with a `viewCount` value of 2048. In this case, a construct such as a semaphore would be used by the first user, P, calling `wait` on the semaphore which is attached to the database, before updating the `viewCount` in the database. When the second user, J, wants to increase the view count as well, they see that the semaphore is raised, and they cannot access the database immediately. Once P is done with the database update, they call `post`. At this point, J is allowed to operate on the database, with a

⁴²e.g. with types and compile-time checks.

```

int sem_wait(sem_t *s)
{
    // decrement the value of semaphore s by one
    // wait if value of semaphore s is negative
}

int sem_post(sem_t *s)
{
    // increment the value of semaphore s by one
    // if there are one or more threads waiting, wake one
}

```

Listing 75: A textbook semaphore description in pseudo-code (Arpaci-Dusseau & Arpaci-Dusseau, 2018)

guarantee of data integrity.

Such a piece of code is interesting for its use of multifaceted engagement with metaphors, its existence between abstract and concrete and its involvement with functional reliability.

A lot of different introductory textbooks, from Dijkstra's original paper to the Wikipedia article on semaphores, rely on analogies in the problem domains to describe the problems implied by a concurrent use of shared resources. Dijkstra refers to problems such as *the sleeping barber* or *the banker's algorithm* as use-cases that are both unrealistic but mentally graspable (Dijkstra, 1965), while the Wikipedia entry refers to *the dining philosophers problem* (Wikipedia, 2023a)⁴³. In this case, we see a macro-level aesthetic device in the form of storytelling, which introduces the reader to the origins and implications of concurrent use in computing systems.

Then comes the use of the term *semaphore* itself. In its mechanical form, a semaphore is a device which signals information to a running train, such as whether the tracks ahead are blocked or clear, whether the train should stop or proceed at reduced speed, or if they should exercise caution. There are multiple properties from this source domain that are applied to the target domain of programming. First, the property of a con-

⁴³A C++ implementation of such a problem can be seen in (Arpaci-Dusseau, 2023)

tinuously running train, whose alternate state is that of waiting, before starting again, and whose state change is dependent on the state change of the semaphore. In the programming context, a thread also assumes a linear continuous execution, and can under certain circumstances be put to sleep, or woken up, by the process. As we get further away from the concept, and closer to the implementation, the aesthetics of the programming expression switches domains, and becomes more fine-grained.

Ultimately, the micro-specific details of the implementation stop making use of metaphors at all and, in doing so, rely on a different kind of representation. The two operations, denoted above `wait` and `post`, are actually left to implementation details. In Dijkstra's original paper, such operations are denoted `P` and `V`, and it is still a matter of debate what those letters stand for, due to the author's use of his native Dutch language (Wikipedia, 2023a). Since these appear as arbitrary marks, we argue that their aesthetic properties in communicating the abstract data type of semaphore changes; they take on the appearance of a single letter: one which is very concrete to the machine, and very abstract to the human.

Openness and transparency

This next example, taken from Adafruit's `pi_video_looper`, exhibits interesting features in terms of openness and transparency, hinting at the reader's implied ability to write.

The program text, published by the Adafruit company, is written in Python and is the source code for a video application running on the Raspberry Pi hardware platform. Both the company and the hardware platform it manufactures are strongly rooted in the ethos of open-source, meaning that it is not just meant to be used, but also to be modified by its users. In this context, the `pi_video_looper` project is made available on the GitHub platform, which facilitates re-use by other users.

The particular section of the program text, whose presentation enables

```

def _load_player(self):
    """Load the configured video player and return an instance of
    it."""
    module = self._config.get('video_looper', 'video_player')
    return importlib.import_module('.' + module,
        ['Adafruit_Video_Looper']).create_player(self._config,
        screen=self._screen, bgimage=self._bgimage)

def _load_file_reader(self):
    """Load the configured file reader and return an instance of
    it."""
    module = self._config.get('video_looper', 'file_reader')
    return importlib.import_module('.' + module,
        ['Adafruit_Video_Looper']).create_file_reader(self._config,
        self._screen)

```

Listing 76: Abstracting hardware specific resources via configuration options in an open-source project. (Dicola, 2015)

understanding for further modifying, concerns two similar functions, `_load_player()` and `_load_file_reader()`, in the `video_looper.py` file, reproduced in 76. These two functions are member methods of the `VideoLooper` class and return the specific video playback processes (such as VLC media player or OMXplayer) and file reading drivers (such as a USB drive or a network filesystem).

To do so, these two methods operate similarly. Based on the current configuration of the running software, they load actual files through Python's `importlib` module, and calls an expected method to return an instantiated object. Since it engages directly with modules in the form of files, rather than through pre-registered abstractions, it gives the end user a glimpse into the workings of the source code as latent scripts of plain text, rather than interpreted code. This might be considered confusing, since this is also the only two occurrences of such a technique in a file that is 500 lines long.

And yet, this architectural choice enables the reader to grasp a couple of fundamental concepts never made explicit otherwise. First, the use of the `_` character prefix for both methods ensure that these are private methods,

and therefore are only directly used in the current class, and not in other, invisible places in the rest of the program text. Second, in a program text whose intention is to be user-friendly, given the number of comments and the culture of the organization from which it stems, an explicit unveiling of dynamic module loading signifies the potential for other modules to be loaded, without having to modify the loading function itself. This expresses the feeling of habitability discussed in 4.3.2, in that readers are invited, in turn, to write into the text and make it their own—here, to use a different video playback or file reading system.

Particularly, the source code is written in such a way that there are hints at the existing parts of the computational environment (the configuration file, the method to be called on the module). This presents a structure in which the writer can insert itself without modifying anything that was not meant to be modified. With three lines of code, each of these methods present an elegant interface between the problem domain (e.g. the media player) and the hardware domain (e.g. `omx` vs. `hello_video`); revealing this loading of files, the program text never states how to add to the source, but rather shows that adding a new playback engine is as simple as writing the playback engine in a new file with at least one specific method as an entry-point (e.g. `create_player`), and changing the configuration file value for the new filename, without having to touch these functions themselves. Furthermore, by acting as this textual location through which multiple computational processes interact, this is an example of a beacon, lighting the way along a non-linear reading process by establishing signposts from which to proceed.

This abandoning of abstraction at a certain level, in order to reveal what should be revealed to a reader-as-potential-writer, builds on a community ethos of hacking, where the machine's workings are laid bare in order to support unexpected changes by unknown individuals. This textual hint at both multiple realities (i.e. how the playback is actually done, inside the `VideoLooper` abstraction) and particular possibilities (i.e. using, or changing

it), creates a particularly welcoming space for newcomers.

Description and execution

To complement our examples of scientist and software developer code, we now look at how source code can evoke a certain sense of the aesthetic by accentuating, rather than reducing, the semantic gap between human and machine.

The poem presented in 77, written in Ruby by Macario Ortega in 2011 and titled `self_inspect.rb`, opens up this additional perspective on the relationship between aesthetics and expressivity in source code. Immediately, the layout of the poem is reminiscent both of obfuscated works and of free-verse poetry, such as E.E. Cummings' and Stéphane Mallarmé's works⁴⁴. This particular layout highlights the ultimately arbitrary nature of whitespace use in source code formatting: `self_inspect.rb` breaks away from the implicit rhythm embraced in *Black Perl*, and links to the topics of the poem (introspection and *unheimlichkeit*) by abandoning what are, ultimately, social conventions, and reorganizing the layout to emphasize both keyword and topic, exemplified in the `end` keyword, pushed away at the end of their line.

From a computer perspective, the program declares a class called `Proc`, a generic and essential construct in Ruby, which has a single member method `in_discomfort?` returning the value of the symbol `me`. The core of the program then takes place in the declaration of the two variables `you_are` and `you`, assigning them a value of a lambda expression. It includes four statements; the first two, `self.inspect` and `break you`, due to their conditions, are never actually executed. The third prints the result of calling `in_discomfort?`, and the fourth recursively calls the lambda expression stored in `you_are` with the argument `you`. Finally, the whole execution of the program is due to the last call to `you` with the argument `you_are`, the

⁴⁴Particularly *Un coup de dés jamais n'abolira le hasard*.

```

class Proc
    def in_discomfort?; :me; end
end
you_are = you =
->(you) do
    self.inspect until true
    until nil
        break you
    end
    puts you.in_discomfort?
    you_are[you]
end
you[
    you_are
]

```

Listing 77: A code poem written by Maca in Ruby. (Ortega, 2011)

symmetric opposite of the last statement of the lambda expression. Functionally, this program text is then a series of recursive function calls.

When read aloud, the poem includes a first mention of the self, before reiterating mentions of you, and inviting tones of uncertainty, through the mention of inspection and discomfort. It thus evokes intimacy, individuality, feelings of absolute, by referring to terms such as true, end or nil (meaning nothing), and short, imperative orders such as do or break. As such, the poem, as read and pronounced by a human, evokes feelings of identity and introspection, felt as negative forces.

The poem also presents features which operate on another level, halfway between the surface and deep structures of the program text. First, the writer makes expressive use of the syntax of Ruby by involving data types. While *Black Perl* remained evasive about the computer semantics of the variables, such semantics take here an integral part. Two data types, the lambda expression and the symbol are used not just exclusively as syntactical necessities (since they don't immediately fulfill any essential purpose), but rather as (human) semantic ones. The use of :me on line 2 is the only occurrence of the first-person pronoun, standing out in a poem littered

with references to you. Symbols, unlike variable names, stand for variable or method names. While `you` refers to a (hypothetically-)defined value, a symbol refers to a variable name, a variable name which is here undefined, and would default to a literal `me`. Such a reference to a first-person pronoun implies at the same time its ever elusiveness. It is here expressed through this specific syntactic use of this particular data type, while the second-person is referred to through regular variable names, possibly closer to an actual definition. It is a subtlety which does not have an immediate equivalent in natural language, and by relying on the concept of reference, hints at an essential *différance* between `you` and `me`.

Reinforcing this theme of the elusiveness of the self, the author maca plays with the ambiguity of the value and type of `you` and `you_are`, until they are revealed to be arrays. Arrays are basic data structures consisting of sequential values, and representing `you` as such suggests the concept of the multiplicity of the self, adding another dimension to the theme of elusiveness. The discomfort of the poem's voice comes from, finally, from this lack of clear definition of who `you` is. Using `you_are` as an index to select an element of an array, subverts the role suggested by the declarative syntax of `you are`. The index, here, doesn't define anything, and yet always refers to something, because of the assignment of its value to what the lambda expression `->` returns. This further complicates the poem's attempt at defining the self, calling the reverse expression `you_are[you]`. While such an expression might have clear, even simple, semantics when read out loud from a natural language perspective, knowledge of the programming language reveals that such a way to assign value contributes significantly to the poem's expressive abilities.

A final feature exhibited by the poem is the execution of the procedure. When running the code, the result is an endless output of print statements of "me", since Ruby interprets an undefined symbol as its literal name, as seen in 78.

The computer execution of the poem provides an additional layer of

Listing 78: The executed output from 77

meaning to our human interpretation. Through the assignment of `you_are` in an `until` loop, the result is an endless succession of the literal interpretation of the symbol `:me`, the actual result of being in discomfort. While we have seen that a symbol only refers to something *else*, the concrete output of the poem evokes an insistence of the literal self, exhibiting a different tone than a source in which the presence of the pronoun *you* is clearly dominant. Such a duality of concepts is thus represented in the duality of a concise source and of an extensive output, and is punctuated by the ultimate impossibility of the machine to process the accumulation of these intertwined references to *me* and *you*, resulting in a stack overflow error.

We now understand that the undefined symbol *me* is to be taken literally, while the output of the program is the result from the mutual recursive calls of `you[you_are]` and `you_are[you]`, creating an infinite mantra (*you are you are you are you are you are you*, etc.) which is heard first by the computer, and only viscerally understood through the execution of the program. Ultimately, an additional theme of the poem can be deciphered: through recursion, the entanglement of individuals depending on each other leads to a semantic and computational overload.

The added depth of meaning from this code poem goes beyond the syntactic and semantic interplay immediately visible when reading the source, as the execution provides a result whose meaning depends on the co-existence of both source and output. Beyond keywords, variable names and data structures, it is also the procedure itself which gains expressive power: a poem initially about *you* results in a humanly infinite, but hardware-bounded, series of *me*⁴⁵.

A final idea here is that the writing of code is an artistic enterprise, both in its traditional understanding of craft, and its contemporary understanding of art. The emphasis on executable code reveals aesthetic possibilities of source code as a medium, in which form, content and func-

⁴⁵Another productive comparison could be found in Gertrude Stein's work, *Rose is a rose is a rose...*, drawing expressive power from the phenomenon of semantic satiation.

tion are closely aligned. Poems such as `self_inspect.rb` are fascinating because they are variably accessible and inaccessible to readers, a function of their readers' knowledge of programming languages and facility with poetry. They also provide means of expression in multiple ways: the visual impression of the code on the page, an aural dimension if read aloud, and the output rendered by the code when compiled. Their possibilities for interpretation, then, are fragmentary, requiring negotiation on these many fronts to appreciate and understand. (Risam, 2015).

If code poems are not immediately functional in the industrial sense of the term, they are nonetheless dependent on the functioning of the program that they describe for a part of their expressive power. This computational function is therefore always a part of the meaning of a program text.

We've seen through this section that the expressivity of program texts rely on several aesthetic mechanisms, connected in a spatial way between a metaphorical understanding of humans and a functional understanding of machines. From layout to double-meaning through variables and procedure names, double-coding and the integration of data types and functional code into a program text and a rhetoric of procedures in their written form, all of these activate the connection between programming concepts and human concepts to bring the unthinkable within the reach of the thinkable. While these techniques are deployed differently according to the socio-technical environment in which the program text is being written and read, they nonetheless all contribute to facilitating the navigation of the program text, be it at the same level of abstraction across parts of the text (such as in 76, where the patterns of writerly text exists across the codebase), or at different levels of abstraction in the same locations (such as in 69, where the syntax abstracts away the unnecessary signifiers of parallel computing).

Ultimately, these aesthetic manifestations of source code in a program

text are all tightly coupled to the execution of that program text. The next section concludes this research by assessing the relation between such function(s) of a program text and its aesthetic manifestations.

5.3 Functions and aesthetics in source code

Through our comparative study of source code with architecture and mathematics, we have seen that aesthetics are not unrelated to ideas of function, or purpose. In the case of architecture, an aesthetic appreciation of a building can hardly be made completely independently from the building's intended function⁴⁶ while, in the case of mathematics, aesthetics are closely associated with an epistemological function. In turn, the aesthetics of source code have been shown to be closely connected to different kinds of cognitive engagement, from clarity to obfuscation and metaphorical evocation.

Each program-text that we have examined in this work always implies a necessity of being functioning in order to be properly judged at an aesthetic level, but the diversity of practices we have pointed to also seems to suggest different conception of functions. In developing the different ways that the function of a program text can be considered, we argue for a dual relationship between function and aesthetics in source code. First, the function of a program text is integral in the aesthetic judgment of such text, both because the status of software as crafted object makes its function the deep structure that is manifested in its surface, and because the standards by which it is judged depend on socio-technical contexts

⁴⁶Larry Shiner explains the limitations of a purely formalistic stance when bestowing an aesthetic judgment on a work of architecture: "*a formalist critic who ignores functions in judging a work of architectural art will be in danger of misjudging it by treating it solely in terms of its purely formal aesthetic properties. For example, if a critic were to judge Gehry's Bilbao museum from a formalist perspective, for example, the critic might have to fault the more traditional looking galleries as out of keeping with the sculptural forms of the rest of the museum, and blame Gehry for failing to unify his design's sculptural form*" (Shiner, 2009).

in which the program text is meant to be used. Second, the aesthetics of source code are not autotelic; rather, the function of aesthetics themselves is to communicate invisible information to the reader and writer.

We first provide an overview of investigations into the relationship between function and beauty. This will highlight how function can be understood from the perspective of intention, but also from the perspective of reception. We then examine specifically the different functions that source code can support, and how they might vary with the context in which the program texts exist. Beyond the computer as the most direct context defining whether or not a program text functions, we highlight how epistemic and social concerns further modulates the technical context in which the function of a program text is assessed.

Ultimately, we conclude on the extent to which aesthetics can be said to hold an epistemic function. Starting from the specificities of source code as a collaborative text with ambivalent semantics, we connect aesthetic value to ethical value through the communicative function, and thus transpersonal requirement, of program texts.

5.3.1 Functional beauty

Traditionally the main focus of aesthetics tended to be on works of art, human creations which were understood to be belong to a specific field, separate from practical concerns. The self-reference of artworks, heightened in the modernist stance of decontextualization and of appraisal of phenomenological form over content, tended to sideline the purpose of such artworks: whether emotional, social, political or epistemological. Nonetheless, proponents of a more conjunctive approach, in which function and beauty are not mutually exclusive, addressing how the function of an object can be perceived through an aesthetic stance.

In order to assess how function can influence an aesthetic experience, we must first identify what constitutes the function of an object. As a start-

ing point, one can relate it to the actions that are enabled by its perceptible properties. Such properties can be designed, and thus actions suggested, by the creator(s) of the artefact, in which case the a-priori intention defines the function, or they can be defined by the user(s) of the artefact, suggesting a more pragmatic approach. Developing this action-oriented, rather than ontological, approach to artefacts Houkes and Vermaas state that *an artefact function is any role played by an artefact in a use plan that is justified and communicated to prospective users* (Houkes & Vermaas, 2004). From this, it follows that the use of an artefact can be either the intention of its creator(s) or the action of its user(s), informed by the typology from which such artefact depends⁴⁷, communicated to its users. These two ways to identify the function of an artifact thus implies that there is some hierarchy of functions: while it is complicated to point out a single proper use for an artifact, some of those uses can be considered more proper than others, based on the interplay of social agreement, ontological status of the artefact, technical properties and individual intention(s).

Given that function is a combination of what is intended by the creator and the effective use by individuals, and that such function is informed by the perception of sensory cues, we can say that aesthetic play a role in understanding what an object does. This perceptual judgment on the use of an artifact is particularly salient in the identification of such an artifact's affordances, visual cues helping a user determining which actions are possible (Norman, 2013). As we have seen the extent to which the aesthetic judgments of source code depend on a certain level of knowledge and skill, we can now investigate the complementary question: to what extent does the function of an artifact influence our aesthetic judgment of this artifact.

A first connection between function and appearance is elaborated by Socrates, and analyzed by Parsons and Carlson, as they emphasize the con-

⁴⁷For instance, the commonly accepted function of a house in architecture, a proof in mathematics or a story in literature.

cept of *fitness* (Parsons et al., 2012). Fitness, here, is the degree to which a collection of features of an artefact minimize the amount of effort or energy spent in order to achieve the goal of a particular user in a given context—that is, to fulfill a function. Specifically, Parsons and Carlson build on this approach by arguing that beauty can be understood as looking fit for function, thus reconciling both appearance and purpose. In their view, aesthetic appreciation can be dependent on the function of an artefact insofar as such fitness can be perceived by the senses. Furthermore, while knowledge of function might not be a pre-requisite for aesthetic appreciation, we can nonetheless note that knowledge indeed factors in in aesthetic judgment, by providing a standards against which achievement can be measured⁴⁸. This implies that there are functional reasons for the perceived appearance of a work, especially of these kinds of work which are expected to primarily perform functions, and then secondarily exhibit positive aesthetic features.

As we developed in 4.3.1, architecture is an artistic field where function is of prime importance, a feature which applies to software development when considering architecture as a metaphor for writing (good) code. However, the exact function of buildings is far from being universally agreed upon for every building: a function might be to provide cheap, temporary housing or to recreate the feeling of Victorian-era luxury, or to optimize its space to allow for industrial production to take place within it, or to present volumes and decoration in such a way that it elicits religious fervor.

In one of the most popular takes on function in architecture, Louis Sullivan, on his essay on the relationship between form and function, develops a concept of function which results from the straightforward intention of the building designers, with respect to the techniques and materials used.

⁴⁸The knowledge of an artwork's function subsequently influences our aesthetic judgment: "*By employing a category in perceiving an artwork, then, we implicitly impose a kind of structure upon its various perceptual qualities*" (Parsons et al., 2012)

In his essay, he applies the adjectives sincerity, honesty, and authenticity: such terms connotate a certain idea of ontological straightforwardness, rather than adaptation to other factors such as effective use or environmental constraints. In this sense, functionalism poses function as a requirement from which aesthetic value follows; an aesthetic judgment has to take into account the intended use of the building.

And yet, such an approach does not take into account the multi-faceted functions of a building. Existing at the overlap of multiple intentions and uses, buildings present a variety of practical functions which in turn influence the aesthetic judgments that are expressed related to it. An example of these varying functions and the accompanying aesthetic perceptions: a library can be judged on how well it conserves the books, on how well it handles sound propagation, on how well it enables the focus of its dwellers, on how well it represents its cultural symbolism, or on the carbon footprint of its construction and use. Each of these will be weighted differently according to whether one takes the perspective of the architect, the client, of the reader, of the employee, or of the passer-by.

To this strict functionalism as a school of thought, one can answer that the goal of architecture is not simply to design buildings, but to *create a conception of living* (Graham, 2000). The function of a building is then to support other functions, or other behaviours of individuals as they engage with a building. Shiner notes how Robert Stecker offers a useful distinction of architecture as artform, and architecture as medium (Shiner, 2009), with the first having a function defined by the creator, and the second having a function defined by the user. In turn, the aesthetic value of the built artefact depends on the conception of living that an individual ascribes to it, and as it uses it. In the case of architecture, one might settle the question of aesthetic value of a building not by stating that it is because is functional, but rather because it is *useful*, involving both the situated use and the situated intent.

Similarly, in the case of literature, the function of a written work is not

just to be decipherable by one other than the other. Starting from an extended conception of literature (Gefen & Perez, 2019), one can look at artefacts such as press releases, novels, poems, and legal codes as fulfilling different functions (such as informing, immersing, evoking or specifying⁴⁹). A well-functioning novel will be thus be judged on different criteria than a well-functioning legal code, and so will a piece of investigative journalism, even though all will share the assumed requirement for the artefact to be functioning with respect to its ontological type—here, that it is a system of linguistic signs decipherable by others. As such, the audience of the work both establishes the usefulness of such work for them and, in turn, express an aesthetic judgment depending on how useful the work is to them, whether it is from an informative, poetic or emotional standpoint.

An object is thus functionally beautiful to the extent that its aesthetic properties contribute to its overall performance; the functional beauty of an object enhances its fulfilling its primary function. In the case of a poem, the primary function might be to evoke emotions, or denote concepts that are seldom explicated in prose; in the case of a novel, it might be to offer a rich storyworld, an engaging intrigue, or a deep portrait of human nature. Here, we therefore do not assess function in immediate, physical terms, where a concrete consequence is needed in order to assess whether a function has been performed. Carlson and Parsons support this view through Benedetto Croce's conception of aesthetics as a linguistic system, in which he defines beauty in terms of expression, by which he means a particular mental process which results in a clear, distinct and particular idea (Parsons et al., 2012). Since communication between individuals can be said to have succeeded or failed, then aesthetics as a communicative process also

⁴⁹Nonetheless, an artefact can function well, and thus be well-judged aesthetically on multiple functions at the same time. The first sentence of the first paragraph of the first article of the German *Grundgesetz*, "Die Würde des Menschen ist unantastbar." (Deutsche Bundestag, 2022), translated as "The dignity of humans is unviolable.", functions both as a fundamental ground for jurisdiction, and as a condensed narrative evocation of the history of the country in the 20th century.

inherit this function of expression. For instance, our discussion of mathematics in 4.4.1 has suggested that the aesthetic judgment of mathematical artefacts, such as proofs, have for functions the facilitating the expression of mathematical concepts, or the acting as a heuristic in order to construct an intellectually satisfying proof.

Both from a philosophical and from a practical standpoint, aesthetic judgments are therefore not independent from the function of an artefact. By developing the concept of looking fit for function, we have shown that aesthetic judgment can be relatively dependent on the function of an artefact. In turn, we have seen that the concept of function itself can be ascribed by the intent of the creator, but it can also depend on the socio-technical context in which the object exists, and from which one poses the aesthetic judgment, involving both human stances and technical properties. An aesthetic judgment of an artifact therefore integrates its perceived function, as well as how well the function is fulfilled.

Since function does not necessitate physical action, we can now look more specifically at the functions which are ascribed to source code. After having highlighted the ontological function of source code as embedded in a computational ontology, we then show the other, pragmatic and paradigmatic functions of source code, and develop on how the aesthetic value of program texts are thus related to such functions.

5.3.2 Functions of source code

From a computer science perspective, a function is a program unit which produces an output for each input. While there might not be some explicit value given as an output, a function in a computer program is nonetheless an action which has the ability to modify some internal state of the machine—that is, they are *effective* (Abelson et al., 1979), and they are commonly expected to provide a tangible change as a result of their execution.

In order to fulfill this definition of function, a given program text must

be correctly machine-readable. This means that its syntax must be correct, before the operations described by its semantics can be executed, and then assessed in the light of its intended result. Rather than looking at function from an ontological perspective, through which all programs that are syntactically correct satisfy the criteria of function in order to support an aesthetic judgment, we can highlight three different modes of valid function. The first concerns its syntactical validity: the syntax expressing what the program text does is correctly parsed by the computer, and can be qualified as a potential function. The second concerns what the program text does, through its operational semantics, and will be referred to as its effective function. Finally, the operation of the program text is compared to what it is supposed to do, according to the programmer(s) who wrote it; this assessment refers to the intentional semantics of a program. As such, a program text can be considered according to at least three different criteria: it is functional because it has a correct syntax, it is functional because it performs a set of computational operations, or it is functional because it has been given a particular intent.

While we refer to the function of a program as the correlation of syntactical validity, operational function and intentional function, each of these subsequent functions can nonetheless be considered as a valid function on their own. For instance, a syntactically invalid program (perhaps due to it being written in an older version of a programming language) might still hold an intentional function (as an educational example). Furthermore, one can also look at it from a more practical perspective, that of *utility*. In this case, the functionality of an artefact is assessed on its use, by situated beings facing precise problems, rather than on its essence.

Moving away from a discrete understanding of function through syntax-correctness (does it function effectively or not?), we can then move towards a more continuous appreciation of how well an artefact functions (is it useful?). If fitness to function can be a principal criteria for aesthetic judgment, the concept function nonetheless remains multifaceted.

Indeed, program texts emerge very quickly from a strictly computational understanding of function (input-process-output) to suggest other possible functions based on the social and technical contexts for interpretation. Rather than being mutually exclusive, we show how the social and the technical influence each other in providing a backdrop for aesthetic judgment through perceived and intended function.

Programming languages have a syntax and a semantics, but they also have a pragmatics—a contextualised purpose within which they are deployed. This concrete, contextualised function can be one of correctly achieving a task, of achieving it within a particular resource constraint, of demonstrating how a task could be roughly achieved, demonstrating one's knowledge of the technical environment in which the task is executed, demonstrating the expressive limitations of such platform, or even challenging readers' assumptions. In all of these cases, function as achievement of intended effect remains an essential aspect through which the quality of the program text artifact can be judged.

The first and more widespread function of a program text in its written form is to be understood. As an essentially technical artefact achieving a particular result through its execution, one is not expected to read source code for entertainment, or out of boredom. Rather, a program text whose formal arrangement enables the construction of useful mental models enabling further programming activity can be said to be fit for its primary function. Nonetheless, this understanding is not exclusively focused on changing, or fixing the program text that is being read, but can have other implications, thus involving different standards for aesthetic judgment.

For instance, achieving a task such as the demonstration of the properties of a language is not the same thing as using such language in the most technically-efficient manner; the first function would be a demonstration of pedagogy, while the second would be a demonstration of skill. As an example of pedagogical program text, the listing 79 shows how the first function declaration is being explicit in its use of conditional state-

```

function draw() {
    if (mouseIsPressed) {
        fill(0);
    } else {
        fill(255);
    }
    ellipse(mouseX, mouseY, 80, 80);
}

let draw = () => {
    mouseIsPressed ? fill(0) : fill(255)
    ellipse(mouseX, mouseY, 80)
}

```

Listing 79: Different ways to write a JavaScript function in different functional contexts, with either a focus on pedagogy or skill.

ments, and conservative in its use of the `function` token for function declaration. These particular aesthetic choices represent the function of this program text: the communication of the basic actions of drawing in, focusing on `fill` and `ellipse` as points of interest for the reader. Conversely, the second function declaration focuses more on the technical context and idiomatic knowledge of the reader by using the ternary operator as a conditional statement, and by using the `let` definition for the function declaration, thus signalling a more technically-oriented function of compressing this color filling operation, for instance to not draw attention on it within the context a larger program, as well as to run on the most recent versions of the ECMAScript standard to which JavaScript conforms.

What is displayed here is therefore judged differently, depending on the intents of the creator and the expectations of the user. Considering an educational function of the program the first function declaration might be judged as more pleasing than the second, which could be considered too cryptic for a beginner. While not exclusive, the technical context in which the program text is written, read and executed—the hardware, programming language and development environment—remains both a determining context in terms of syntactical and semantic fulfilling of functions, but

also one where artistic creativity supports a pluralistic conception of function.

Particularly, hackers (see the description of the community of practice in 2.1.2) display a practice of writing and reading source code which plays with the naïve definition of technical function of achieving in a straightforward manner a standard task. The aesthetic standards of hacking involve a valuation of materiality and epistemology, demonstrating understanding through obfuscation. The knowledge of the machine is implied through this dimension of exclusiveness: being able to do what others do not know how to do. The function of such hacker code is thus not to be easily understood, but rather to offer a convoluted and engaging puzzle as to how the effective function of a program is even possible, given an obfuscated syntactical function.

Practices of hacking thus tend to deviate from the original, intended function of a given hardware platform or software system, in order to find a new function. For instance, in cases of XSS attacks, in which an input field on a website is exploited not to provide information to the website system, but rather to penetrate and occupy it. From the website administrator's perspective, the function of the website is perverted, and the code which performs such function might be considered as ugly whereas, from the hacker's perspective, the function of the website as a data input is actually sublimed by circumventing arbitrary protections put in place to prevent certain types of data to be injected.

By switching the paradigms of what an artefact can do, or should do, hacking takes on an epistemological role, providing new knowledge about the possibilities of an artefact, and displaying material proof, in the form of an executed output and a program text. Indeed, hacking relies not on the proclamation of skill, but on its execution: it is the combination of the intended action of the writer and the effective operation of the computer which grants to the function of the artefact a particular quality. This is purposefully at odds with hacker syntax, which tends to be more obscur-

ing than enlightening. It is this unexpected ability to perform the intended function while not revealing its function through syntactical means which in turn informs aesthetic judgment. Here, the aesthetic judgment is directly informed by the relation between written syntax and executed action, and such an executed action affords intellectual engagement with understanding how such action is even possible. This epistemological approach, of understanding how such a hack can be done, thus relies on a visual component: such component of the hacking aesthetic is the *visual tension*, the fact that an object might not look the part, but actually is doing such part is reflected in practices of obfuscation and obscurity. In cases of obfuscated code, being able to hide visually the purpose of a program text while conserving its invisible, technical-functional properties⁵⁰ is thus considered a virtue, and suggests a positive aesthetic judgment.

Hacking allows us to consider the aesthetics of dysfunction, in which one can find pleasure in addressing, and then subverting, the original function of the artefact. An example of this subversive approach can be seen in the design and use of *esolangs*, esoteric programming languages whose sole purpose is to exist as playful perspectives on the arbitrariness and yet unescapable exhaustivity of computational syntax. Rather than being usable in practical situations and thus useful, they rather function as a commentary on the material of source code, displaying the extent to which something else can be a programming language. For instance, Daniel Temkin's *Folders* language, seen in 5.1, uses directories in order to perform computational operations. While this might not have any reason for existing in a productive environment, this nonetheless performs the function of expanding one's understanding of what is a programming language, at a more fundamental level than everyday software development. There can be effective function separately from utility: here, the extent to which some programs seem to stray from practical function has itself the epistemological function of enriching our understanding of the nature of

⁵⁰For instance, in the cases of Perl one-liners or obfuscated C code

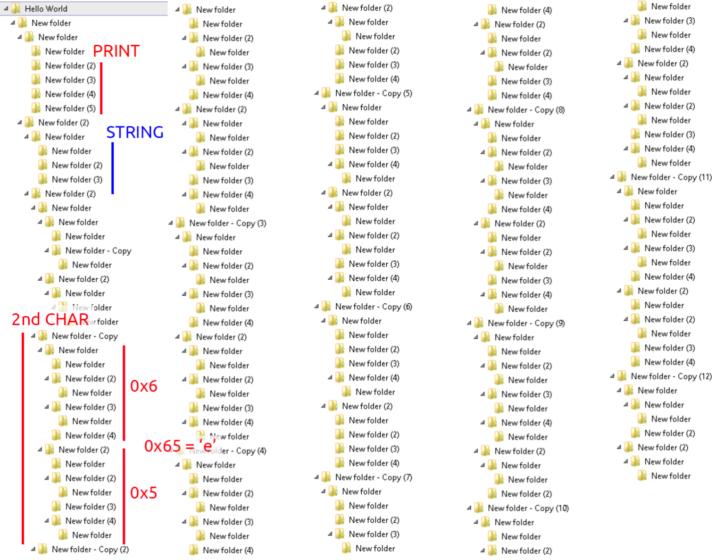


Figure 5.1: Implementation of the traditional "Hello, world!" program in the Folders programming language. (Temkin, 2015)

computation.

If hackers subvert our conception of what it means to read, write, execute and understand source code, so do code poems. They are the complementary opposite of such material investigations, arguing rather for the human expressiveness of programming languages. Here, the function is not effective—what actions does programming enable—but useful as it enables new, personal understandings of concepts rendered through programming—what thoughts does the program text enable.

The use of programming languages for poetic purposes (such as 22 or 77) provides a reconciliation of machine execution with human interpretation. Here, the technical environment provides a requirement of syntactical and operational function, while the social environment provides a frame of understanding, reliant on the intention of the writer, of the reader, or of both.

Software can hardly be separated from what it does, and yet cases do exist in order to focus on under-examined aspects of how an artefact is tradi-

tionally thought, subverting a naïve understanding of function. In architecture, particular structures do escape expected functional assessments in order to provide new understandings and new possibility, as in the construction of Renaissance follies (displaying symbolic value) or the modern pavillion (displaying technical prowess). Neither of these constructions abide by the traditional function of architecture (sustaining and sheltering life) but, precisely, these are exceptions confirming the rule, and these extreme examples also help to highlight the default, expected standard, often taken for granted.

We have shown both the variety and necessity of functions in the aesthetic judgment of program texts, stem from their nature as technical artefacts, but also rely on social environments for a program text's primary function to be considered. Furthermore, we have shown that such function is not just material, in the sense of what technology can help us do, but most importantly epistemological, in the sense of what the technology can help us think about, whether a problem domain, a skill, or a feeling. We now conclude this chapter by extending from source code aesthetics into other domains of crafted objects in order to suggest some perspectives on function and aesthetics and how they relate to ethics.

5.3.3 Aesthetic and ethical value in program texts

The function of aesthetics in source code is thus an epistemological one, in which particular formal configurations act as a guide towards knowledge of the program's contents, either as a heuristic from the writer's perspective, or as a cognitive scaffolding from the reader's perspective. Furthermore, its existence as a technical object also implies an existence which can rely both the writer's intent and the reader's use. As intermediary objects, program texts thus possess an ethical dimension, insofar as they need to consider both oneself and the other in the making of decisions resulting in a positively-valued result.

One of the specificities of program texts is that they are collaborative and open-ended, particularly in the open-source movement, which tends to make all program texts writerly texts. Since the audience can become the creator of modified functional technical systems, expressive devices also act as communicative devices, and thus take on a relational dimension. The program text acts as a bridge between the intent of the ideal version of the software, the reality of executing hardware, and the mental spaces constructed by readers and writers. Concluding in the *Languages of Art*, Goodman develops on the relationship between art and understanding, as he compares an artistic attitude, involving an aesthetic experience, with a scientific one:

The difference between art and science is not that between feeling and fact, intuition and inference, delight and deliberation, synthesis and analysis, sensation and cerebration, concreteness and abstraction, passion and action, mediacy and immediacy, or truth and beauty, but rather a difference in domination of certain specific characteristics of symbols. (Goodman, 1976).

As a crafted technical artifact borrowing from the characteristics of natural language symbol system, and executed by the machine, source code as a medium occupies a hybrid place between art and science. With a genealogy rooted in hard sciences, and with an ontological nature of functionality, it always involves the concept of correctness, a correctness which is always verified through execution. On the other side, the complexity of the computational systems being described, and the uniqueness of the syntax and semantics offered by the medium of source code that is a programming language, require a certain amount of expressiveness found in the use of metaphors to represent concepts from both the problem domain, and from computation itself. Still, this exchange of knowledge takes place most often between two subjectivities: the writer and the reader.

Writing is, in the moment of its doing, a mostly personal act. When

a programmer writes some source code, they do so in a somewhat intimate manner: the only functional judges are oneself and the machine, while the only aesthetic judge is oneself. Criteria for aesthetic judgment, at this point, include three axes: the accuracy of the action performed once the program is executed, the ability of the program text to express the concept that is being implemented, and the adequacy of this formal arrangement with the problem at hand—that is, the idiomaticity and elegance of the program text as a solution. Indeed, functional entities appear graceful when they are free of features that are extraneous, or irrelevant in relation to their function. This translation of function into appearance in turn depends on the complementary position: reading source code.

Once a program text is written—that is, once a computational representation of the world has been given textual form—the process of reading introduces new constraints for an aesthetic judgment. Reading a program text involves a process of decrypting the realized adequacy between intent, form and function. This amounts to identifying the semantic affordances, under the form of structure, syntax and vocabulary used by the writer(s) to communicate the ideal action of the program. We put here a particular emphasis on the relationality of such expression. The position of the reader always involves an otherness with respect to the position of the writer. Providing an aesthetic judgment from a reader's perspective thus involves establishing the elegance, fitness, and interest of a certain piece of source code, in accordance with the writer's judgment; because the program text is interpreted by a mechanical third-party, the computer, its value is not exclusively decided by subjective perspectives, and it therefore gains in objectivity. Bruce McLennan writes about this objective component of aesthetics in software engineering, as a way to harmonize the endeavours of the different individuals involved in creative work, from practical construction to the devising of new ideas⁵¹.

⁵¹"Like cathedrals and scientific theories, large software projects are the result of the efforts of many people, and aesthetic standards provide criteria by which individual contributions can

This particular relational stance relates to Gerald Weinberg's conception of *egoless programming* (Weinberg, 1998). Considering the practices of professional software developers, Weinberg observes that too much ego lead to non-functional software, as one could not benefit from an external readership in order to weed out mistakes and bugs. One of the ways code can be found to be of good quality is through the giving up of personal ownership for a more collective one. This has first an immediate functional effect, by providing additional layers of quality assurance through the perspectives of everyone which contributes to the program text, directly, or indirectly. It also has an aesthetic consequence, in the form of programming style.

As mentioned in 5.1.3, programming style in its individualistic conception is frowned upon, as style should be understood as a collective agreement of ways of doing things. Marielle Macé, in her study of style as a form of life, considers that style is a negotiation between personal and collective ways of being, as an agreement on what matters and on how it should be done⁵². As she considers style as a way of appearing, a way of doing and a way of inhabiting an environment. This last point ties back to the notion of habitability mention in 4.3.2. Aesthetics in programming, as a positively valued manifestation at the level of the sense and at the level of the intellect involves a form of transpersonal activity and ties back to a moral virtue of providing habitable spaces. An aesthetically pleasing program text in which one feels at ease to operate, and thus abides by a collective notion of style.

All source code aesthetics relate to a certain conception of function, in-

be objectively evaluated" (Schummer et al., 2009)

⁵²"Le « comment » comme lieu d'émergence des valeurs, lieu de querelle sur ce qui compte, lieu d'engagement sur ce qui nous divise et ce qui nous relie. . . C'est la question éthique et politique qui s'ouvre ici, dans sa force d'appel et son indécision fondamentale, celle d'une vie qui est toujours à faire, à débattre, et qui se fonde sur nos différends. Car les manières du vivre n'assument pas un sens ou une valeur a priori ; elles sont le sens et la valeur qu'il y a à faire." (Macé, 2016)

volving technical achievement and interpersonal existence. An aesthetic judgment of a program text is, in this understanding, the judgment of the perceptible manifestations in source codes allowing for the comprehension of a technical achievement according to contextual standards. These manifestations are therefore not just expressive (personal), but primarily communicative (interpersonal), aiming at the transmission of concepts from one individual through the use of machine syntax through the dual lens of human-machine semantics. Indeed, code that is neither functioning for the machine, nor meaningful for a human holds the least possible value amongst practitioners.

In the overwhelming majority of cases of program texts, the expectation is to understand. Writing aesthetically pleasing code is to write code that engages cognitively its reader, whether explicitly for software developers, pedagogically for scientists, adversely for hackers and metaphorically for poets. This engagement, in turn, supposes an acknowledgement from the writer of the reader. The recognition of the existence of the other as a reader and co-author, implies an acknowledgement as a generalized other in the sense that anyone can theoretically read and modified code, but also as a specified other, in the sense that the other possesses a particular set of skills, knowledge, habits and practices stemming from the diversity of programming communities. This stance, between general and particular, is one that shows the ethical component of an aesthetic practice: recognizing both the similarity and the difference in the other, and communicating with a peer through specific symbol systems.

In conclusion, this chapter has argued first that programming languages provide a socio-technical linguistic environment for the writing of program texts. Acting first as an interface to the computer, programming languages, without overly-determining the practice of programmers or the

content of what is being programming, nonetheless influence how it can be said, through idiosyncracies and stylistic devices. This has established the idiosyncratic status of source code as a medium, and its existence between technical and social, expressive and communicative, individual and collaborative.

We then presented a framework for aesthetics of source code, through the dual lense of semantic compression and spatial navigation. To do so, we started from a layer-based approach to the points in which aesthetic decisions can take place in source code—that is, across structure, syntax and vocabulary. Broadening this approach, we then showed how these different levels involve an engagement with semantic layers: between the human reader, the machine reader and the problem domain. The minimizing of syntax while best representing the different concepts involved at these different layers results in semantic compression. A source code with aesthetic value is one which balances syntactic techniques, structural organization and metaphorical choices in order to communicate a socio-technical intent of a functional artefact. In turn, semantic compression supports the shifts from different scales or perspectives the engaged programmer needs to operate as she navigates through her non-linear exploration of a program text.

We concluded this study on the relationship between function and the aesthetic judgment of source code by showing how aesthetic writing involves a certain conception of ethics. Not only is the function of aesthetics in source code is epistemological, in that it enables the acquisition of knowledge, but program texts also involve a tight intertwining between writer and reader. What must be communicated then is not just what the program does, but how it does it within a given socio-technical context. As we reconsidered the status of style in programming as a transpersonal way of doing, this allowed us to qualify source code aesthetics not as a primarily individual endeavour but moreso as a way of acknowledging the other.

Chapter 6

Conclusion

A piece of source code, as the linguistic representation of computational processes, themselves representations of a problem domain, is an ambiguous object. Such an object exists at the overlap of both human and machine comprehension, operates through un-intuitive scales of time and space, and is often hidden away by the executed processes of which it is the source. And yet, source code practitioners, those who write and read code, agree on the existence of a certain sense of *beauty* in program texts.

The research aims of this thesis were to highlight the specific aesthetic properties exhibited by varieties of source code. How does source code beauty manifest itself? Under which conditions? And to what end? Answering these questions, we showed how other aesthetic fields are used as metaphors in the aesthetic appreciation of source code and we identified the role aesthetics play in the existence and purpose of source code—with a particular focus on its role as a cognitive facilitator, and on its relationship to function. Our methodology started from an empirical approach, looking at specific instances of source code, and from the analysis of the discourses surrounding and commenting these instances. From this initial study, we identified several lexical fields that programmers refer to when they evoke the aesthetic properties of source code—literature, ar-

chitecture, mathematics and craft.

Along with this first research axis, we also noted how the aesthetic judgement in source code is closely tied to its functionality. Indeed, any aesthetic value is dependent on the correct behaviour of the source code; ugly code is often related to its apparent bugginess and difficulty in understanding its function, while beautiful code implies that the actions resulting from the source code are conform to what the programmer had intended, along with being presented in the best possible way.

Such a definition of a *best possible way* is dependent on the social, cultural and economic spheres within which the code is produced. These include the social environment of the programmer(s), the technical environment in which the code is run and built, and the problem it aims at solving. Similarly, the concept of function within program texts has been shown to also be multifaceted, includin what the code should do, what it actually does, and how it does it.

The aesthetic properties of source code are therefore those of a semantic representation of computational space-time, whose purpose is the effortless communication of the operations of the computer, the intention of the programmer(s) and the representation of the world. In this sense, aesthetics perform a cognitive function.

6.1 Findings

The rest of this conclusion will address each of our initial research questions' findings, followed by an examination of the limitations and contributions to existing research on source code. Throughout, we will summarize how our comparative approach highlighted medium-specific aesthetic devices whose function is to engage epistemically with their audience.

6.1.1 What does source code have to say about itself?

One of the gaps we identified in source code-related literature is that there was a missing overlap between a broad empirical approach and a robust conceptual framework, expliciting the nature of source's code properties. For instance the works of (Paloque-Bergès, 2009; Cox & McLean, 2013; Black, 2002) establish an overview of source code with explicit aesthetic properties, but rely on a remediating approach to assess source code as a literary-semantic tool, or as a discursive-political object, respectively, all the while focusing on the subset of so-called creative code. We intended to complement this initial work by highlighting source-code-specific aesthetics—that is, formal manifestations with a communicative purpose, beyond a strictly literary perspective.

Starting from trade literature on the topic, and complementing it by cases of close-reading program-texts, we have highlighted both structural and contextual specificities of source code. Building on existing work across disciplines, such as Martin (see 2.3.3), Gabriel (see 4.3.2), Lakoff and Johnson (see 3.3.1) or Détienne (see 3.2.3), we have found several properties which seem to be unique to source code, and supports a conception of source code as a material used to construct dynamic semantic spaces.

First, *conceptual distance* is key at a structural level: correlated expressions, statements or variables that affect or depend on the same concept (e.g. a file operation or a user account), should be located close to one another in the source code. This counterbalances the entropic tendency of source code to tangle itself, such that the reader has to follow the convoluted machine path of execution, rather than the human conceptual grouping of executable statements.

The *conceptual coherence*, and thus its ease of understandability, is also manifested in conceptual atomicity and conceptual symmetry, respectively meaning that a given explicit fragment of source code should only refer to one specific operation, at a given level of abstraction, and

that fragments of source code that do similar things should look similar as well. Also previously identified as *separation of concerns*, these two principles allow for the abstraction of a given syntactic unit by grouping all the statements into a single action or declaration, thus operating as a bridge between human understanding and machine understanding.

At the lexical level, source code is multi-dimensional. On the one hand, it operates on an axis that goes from *global* to *local*, whereby global tokens that are used, and are visible, across the whole application code are very explicitly named, sometimes in all uppercase, while local tokens, whose lifetime does not exceed a few lines, tend to be composed of just a few letters. Here, variable length and cap size is closely related to the concept of *scope*, yet in a slightly looser way than from a strict programming language perspective. On the other hand, lexical tokens can belong to three different lexical fields. These lexical fields are whether a given token refers to (1) an individual meaning, (2) a machine meaning, or (3) a domain meaning. For instance, the names `start_time`, `UTC_UNIX_STRING_NOW` and `meeting_time` might all refer to the same moment in time, yet from different perspectives. The first naming, as an individual meaning, is significant in a narrow context, for a narrow set of individuals at the moment of writing or reading. The second naming is a machine meaning, which refers to how that moment is perceived by the computer. The third is the domain meaning, which is how end-users will refer to that particular moment. The use of a different names to refer to a single entity has also shown that metaphor theory comes into play.

For some, a piece of source code which can choose a token that will balance these three meanings in order to convey these three senses of the value at hand will be considered aesthetically pleasing. For others, writing tokens at the extreme of either of these three poles can be considered as a marker of aesthetic success, accompanied by a certain degree of expertise. For instance, code poets would tend to focus on the domain meaning, in which tokens are only referring to non-computing terms, and evoke

poetic concepts instead. Conversely, hackers share a standard for brevity and directness—by making their tokens as short as possible, e.g. reducing them to bytecode, they strive towards existing as close as possible to the hardware that the code depends on, and therefore display unusual feats of performance.

As source code gets closer to the hardware, the representation of its semantics change. Aesthetics move away from surface and towards depth, and human-readable names disappear. So, while syntax such as names and comments might be beacons with an aesthetic potential, positively-valued structural arrangements subsist in a different form. One form of structure, such as the files and folders organization of some codebases, create a sense of familiarity in the situated programmer, as seen in 4.3.2. However, we can also note that structure can evoke less-human concepts, and be considered aesthetic insofar as they present a stimulating mental puzzle where the discovery of the program text's computational function is the ultimate reward, as discussed in 2.1.2. For instance, the program text presented in 8o displays an aesthetic structure, independent from syntax. It seems at first very cryptic, but nonetheless exhibits a certain regularity and symmetry in its layout.

Published in *xchg rax, rax*, a collection of riddles in the Assembly language, this seemingly cryptic example allows us to show that, while no arbitrary names are used, structure nonetheless survives (xorpd, 2014). Borrowing from poetry's lexicon, we can identify four stanzas, twice of four lines, and twice of a single line. Syntactically, one can easily spot the repeating of a pattern, with a mirrored relationship between `rcx` and `rdx`, two of the CPU's memory registers where temporary information is stored. The first stanza raises a given number to the exponential of itself, bitshifted to the right ($x^{(x>>1)}$). The second increases the original number, the third stanza repeats the first operation with the previous number and the last, concluding line calculates the `xor` result. The 3-1-3-1 repeating pattern and the similar registers being used across stanzas makes it such that, at first

```

mov    rcx,rax
shr    rcx,1
xor    rcx,rax

inc    rax

mov    rdx,rax
shr    rdx,1
xor    rdx,rax
xor    rdx,rcx

```

Listing 8o: This Assembly listing represents a pair of numbers as reflected binary numbers, and then performs a logical `xor` operation on a pair of numbers. The structure of the program text itself, through its symmetry, hints at the patterns exhibited by such reflected binary encoding (xorpd, 2014).

glance, its structure evokes the concept of reflection and symmetry.

Semantically, 8o tells another story. The semantics of this program text is to compute the exclusive OR of two consecutive reflected binary codes. The binary reflected code, also called Gray Code, is a way to represent binary numbers in such a way that incrementing from one number to another only changes one byte. Following this notation, incrementing from 1 to 2 would be written from 001 to 011 rather than from 001 to 010¹.

This kind of binary number representation relies on a linear increment which exhibits further structural properties. For instance, reflected binary numbers are used in electronic and digital communications, as well as signal processing, in order to reduce errors in information analysis (since only one bit changes from one binary number to the next, it becomes easier to trace through linear changes and catch mistakes). For instance, this particular snippet could be used to detect if there was an error in an encoding of information by acting as a test control: since it calculates the exclusive

¹For more details on how this particular program text does its encoding, see (Sanchez, 2016)

or of two consecutive numbers, there should only ever be a single 1 in the result, and any more or less flipped bits would indicate an error in the processing.

However, like the aesthetics of mathematics, we here start from this somewhat simple syntactical representation, followed by a changing of the scale at which it operates, in order to grasp a more complex, yet highly regular, structure. In fact, such a structure is used in puzzles like the Towers of Hanoi, or the Chinese rings puzzle, and is an example of combinatorial algorithms (Knuth, 2011), reconnecting the hacker aesthetic to a certain kind of playfulness. Through a poetry-like layout and with a mathematical intent at evoking complex numerical concept, a seemingly simple program text allows us, with a subset of source code aesthetics, to grasp a complex computational structure. Away from names and human idiosyncrasies, aesthetics persist.

The name Assembly, the language in which 80 is written, also evokes hints of craft, and program texts in Assembly are often referred to as "hand-crafted". As we showed in 2.3.3, with craft comes communities of practice. Such communities are also an influence on what is to be considered aesthetically pleasing code. With a strong ethos of craft running as a thread throughout each of these identified communities (see 2), well-written code is *idiomatic code*. This implies that the reader and the writer both possess some knowledge of the specificities of the language or hardware that the code is being written with and executed with. While skilled work is often related to a positive appreciation of the result, craft also includes a conception of being usable.

This social existence of code and its connection to skilled work also led us to examine the role of *style*. Style, in this case, is valued positively when it represents the acknowledgment of the social existence of code: by choosing style as a group marker rather than as an individual marker, a source code is judged positively based on its altruistic ethical nature.

More fundamentally, the aesthetic properties of source code are de-

rived from a conception of code as a semantic material which in terms is assembled, and apprehended, as a spatial construct, rather than a strictly literary, mathematical, or architectural material. Code *navigation*, code *structure* or code *compression*, are terms which all belong to a lexical field of spatiality, whether visible or not; the aesthetic properties of source code are tightly related to this apprehension and revealing of conceptual spaces constructed from machine-readable lexical tokens represeting problem domains—or, in other words, *thought-stuff* (Brooks Jr, 1975).

6.1.2 How does source code relate to other aesthetic fields?

Aesthetic properties of source code were deduced from an empirical approach. We identified the different lexical fields that programmers referred to as they justified their aesthetic judgments on program texts. Specifically, we have identified how references to other fields of activity were used as a metaphorical device in order to better qualify source code (e.g. "source code is *like literature...*").

Literature acts as a metaphor for source code through the mapping of linguistic tokens as the building blocks of both natural language texts and program texts (see 4.2), while the architecture metaphor includes spatiality and habitability, along with an explicit dimension of function (see 4.3) and the mathematics metaphor works through a mapping on abstract conceptual structures and strive for elegance (see 4.4). We saw that the metaphorical mapping of each of these source domains ultimately reveal and hide particular aspects and affordances of source code.

Literary aesthetics facilitate the comprehension of the scope of variables and of the intent of the programmer in relation with the problem domain. They denote the purpose and intent of specific values, expressions, declarations and statements in a natural language, with a potential both for poetic evocation, cryptic obfuscation, or plain misinterpre-

tation. Despite Yukihiro Matsumoto and Donald Knuth's statements that writing source code is a literary art (Knuth, 1984; Matsumoto, 2007), this turns out to only be partially true: the most literary parts of source code—comments—are also the parts that are the most decoupled from the actual source code, and are entirely invisible to the machine.

A strictly literary understanding of source sets aside the particularities of the reading process of source code and the temporal control of the writer. A traditional, natural language literary work will assume a linear, front-to-back readership, while source code is defined by its potential ability to jump from any part of the text to any other part of the text. Given this radical difference, references to architectural aesthetics help to establish structural patterns of familiarity and spatiality. Even though it does not operate on concrete, "natural" space, the quality of the disposition and combination of the application components on the source code page enable a better navigation of the source code's conceptual space. Furthermore the metaphor of code as literature also hides the differences in authorship: literature often assumes a single author, while code is in majority written collaboratively, in such an intertwined way that it is complicated to attribute the origin of program texts to a single person (as in the tracing of the authorship of 13), a complication which increases with time and the modification of program texts.

This reduction of a vast conceptual space to natural language representations, and presented as clear, delimited set of interrelated components reveals the tension in source code between form, function, and the fundamental concepts of computation. In this respect, mathematical aesthetics enable the condensation of knowledge and insight in the least amount of tokens, minimizing noise, and related to poetic expression. Particularly, this ability of representing complex ideas into simple terms is a process of *compression* shared across poetry, architecture and mathematics, and resulting in an elegant structure.

The architectural metaphor of source code further confirms this struc-

tural aspect nature of source code. In architecture, a building ultimately enables flows of people within a static configuration. Similarly, one can consider source code as the static structure within which the dynamic processes of computation are executed, as illustrated by the term *control flow* or *leaky abstractions*. In a sense, then, source code can be considered as the blueprint of software, just as a floorplan can be considered the blueprint of a building—even if such floorplan, in this case, would need to be at the 1:1 scale. Structure for computational processes, then, but also structure for humans. As discussed in 4.3.2, the formal arrangement of source code which enables a programmer to inhabit it, to feel at ease in reading and modifying such source code is also positively valued. The structural metaphor of architecture thus works at these two levels.

The maxim *form follows function* emanates from the field of architecture and therefore allows us to highlight the requirement of function in the definition of source code aesthetics. Software needs to be functional in order to be aesthetically judged, and aesthetics facilitate the programmer's understanding of what a program text's function is. This functional aspect also corresponds to a distinction between the essential and the superfluous or, in architectural terms, between the decorative and the load-bearing. In both architecture and programming, there are arguments being made for the decorative, as a communicative device for a human touch, while the load-bearing element maps to the elegant engineer, the rather impersonal construction which can nonetheless do the most with the least.

Finally, thinking of code as architecture allows us to highlight the notion of craft in the appreciation of well-written source code. Software craftsmanship is both an approach to detail as a particular relationship to material, tools and knowledge. It is a pendant to an overall architectural structure in which a bird's eye view of folder, files, variables and function declarations can provide a grasp of the overall arrangement and style of the software described by the program text. At the micro-level, an architectural approach to source code raises the question of its status as mat-

ter to which one can shape into functional structures. The carefully assembling of a program text, by programmers as craftpersons, ultimately reveals the materiality of source code as a medium. A crafted program text takes source code as a material; a cognitive material, but a material nonetheless, a kind of *thought-stuff*. The attention to detail, superfluous for the amateur practitioner, nonetheless communicates a certain kind of know-how (see 3.1.2) in the places where one can express their individuality, or focus on a more impersonal and altruistic approach, thus displaying a deep understanding of what they are doing.

This cognitive element is further revealed by the mathematical metaphor. The most obvious connection is through the common use of a formal syntax in order to express complex concepts. While initially terse and foreign, such a language enables a certain kind of play. One can reduce an expression, replace its terms, consider problems from a different angle, at different scales, under different conditions. etc. This play with symbols reveals a certain malleability and modularity of its object, and further supports our approach of code as a cognitive material. As shown in 2.3.2, aesthetics in source code, as in mathematics, can be seen as both a by-product and a goal to be reached, implying a certain ideal formal configuration of symbols for a given problem. Conversely, this relationship with cognition also operates at the earlier stages of writing code: as a heuristic, a positive aesthetic judgment on a work-in-progress leads the programmer and the mathematician alike in the right direction of a correctly functioning program text or demonstration.

Most visible in the hacker aesthetic 2.1.2, code as mathematics makes obvious the relationship of aesthetics with intellectual engagement. Whether it is to understand certain subtleties at the algorithm design level, at the programming language use level, or at the hardware configuration level, aesthetics have the function of communicating the author's knowledge to the reader, either by making the syntactic representation the simplest possible, while not compromising with the integrity of the underlying

concepts or by making this representation so obfuscated that these formal arrangements announce a pleasurable brain-twisting puzzle. In any case, the aesthetic experience of code, just like the aesthetic experience of mathematics is not one which relies on immediate, emotional reaction. Rather, it demands from the reader a focused attention and cognitive abilities of modelling the space time of a program text; in turn these two requirements are impacted by formal arrangements, making concepts harder or easier to grasp.

Aligning with the conceptions of code as literature and code as architecture is that of *elegance*. We defined in 2.2 the notion of elegance, from poetry to engineering, as the ability to do the most with the least. Mapping these aesthetic metaphors onto source code confirmed that a program text written in a way that uses the minimum amount of required tokens in order to perform the fullest version of its function is one of the most praised aesthetic abilities. Robust, sparse and straightforward program text is considered a beautiful achievement, one in which function, structure and skill are intertwined to produce the most with the least. Here, this definition of "the most" is not only one based on quantitative performance such as CPU cycles, but also on its easing of the cognitive burden in understanding and engaging with the technical object that is source code.

However, what the mathematical metaphor does not show is the relationship between elegance and context. What "the minimum required" and what "the fullest version of its function" depend on various factors, from external technical requirements, programming language, number and skill of collaborators, etc., something which mathematics, in its presentation as a *lingua universalis*, sets aside.

Overall, then, the overlap of these metaphors have led us to identify two main aspects: semantic compression and spatial exploration. Semantic compression concerns the ability of a notation to express complex concepts through quantitatively and qualitatively simple combination, while spatial exploration concerns the ability of source code to be structured in

such a way that is both evocative (the broad shape of things have a relative connotation to what these things can do) and sustainable (the structuring of a function ensures that a given action will not have unexpected side-effects), with the ultimate purpose of facilitating the navigation of program texts by the programmer. Furthermore, rather than being opposites of one another, each reference contributes to the purpose of source code aesthetics by clarifying the structure of the code at multiple levels and dimensions.

Ultimately, all of these elements thus relate to communication and cognition, and to how the (invisible) purpose and intent of the code can be communicated in (visible) lines of a language straddling the line between machine and human comprehension. Literature, architecture, mathematics and engineering all rely on a set vocabulary to enable through understanding; their efficiency at doing so can be assessed by the reader's correct or erroneous estimation of what are the fundamental concepts of what is being communicated to them. Keywords, tokens and beacons are all elements which have been found to structure the writing and reading of source code, allow the programmer to establish a cognitive map of the abstract structure of the program text.

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

This final correlation of aesthetics with the communication of intent and purpose now leads us to address our third research question: the connections between form and function in source code. We have shown that, in the case of software engineers, aesthetics can be used to facilitate understanding in a functional context, or that, in the case of hackers, aesthetics can be a display of a deep understanding of the material at hand. As for scientists and poets, aesthetics perform a role of compression of complex concepts (be they scientific or poetic) into a concrete form. Aesthetics are

both conditioned to, and signifiers of function.

However, the most crucial aspect of the aesthetics of source code is that they any positive evaluation is negatively affected if the executed code does not perform as intended, such as if there is a mismatch between what the original programmer(s) intended, and how the actual machine behaves. There is very little guarantee of such a synchronization: the programmer might say something and the machine do something different, and it is not *clear* what or where exactly is that difference. In this case, the program text, as the only component of software taken into account by the computer, is also the only canonical source of investigation into fulfilling the functional nature of the program.

In this sense, the quality of an aesthetic property (e.g. consistence or coherence) can be judged on whether it adequately represents a given concept, behaviour or intent. The unique aspect of this aesthetic judgment of source code is that there are indeed two judges: the human(s) and the machine, whereby the possibility for human assessment is dependent on a presupposed machine assessment. In all the different groups of writers identified, *correctness* always conditions *pleasantness*.

This is verified only to a certain extent for poets, whom do not require a program text to be productive in order to be given an aesthetic value. Still, in the case that the poet does write a syntactically correct text from a machine perspective, and a semantically evocative text from a human person, the artistic quality of the work created emanates from this technical feat. Traditionally, the effect is similar: by respecting particular formats (e.g. the sonnet, the haïku or the alexandrine verse), the poet displays technical virtue along with emotional sublimation. One understanding of the poem's functioning (rather than its producing) is how each chosen words manage to fulfill the expectation of the technical form and the evocative content. One might say that a particular choice of word or line "works" at a certain location but "doesn't work if moved at another part of the poem.

Poems also perform a kind of function that is not as immediately pro-

ductive as a database query. By joining the technical and the emotional, they perform a more symbolic function about the space of possibilities and the space of the thinkable given to humans. The poet's dual display of skill relates to a conception of art as a connection between the technological and magical highlighted in subsection 3.2.1. Displaying artistic creativity within source code can thus be seen as a way to enchant the technology of software, by representing it as a technically excellent crafted object, imbued with poetic expressivity.

This tight coupling of function and appearance, something already very present in architectural aesthetics (see section 4.3), also echoes with Nelson Goodman's theory of art as composed of a language system used to express complex ideas (see 4.1.1), and practices of craft and toolmaking (see 2.3.3). Source code, while remaining subject to function, nonetheless allows for a certain versatility in the expression of the concept (ranging from explicit to implicit); in turn, this expressivity depends on a given level of skill and practice in the idiosyncrasies of the programming languages used and the programming communities in which the source code is written (see 5.1.3). The proficiency in a language involves a "right way to do things", resulting in "things looking good", and hints at the fact that there is a certain level of expertise needed to assess the quality of the aesthetic properties of a program text, and that the novice cannot be expected to provide an informed aesthetic value judgment.

6.2 Contribution

Overall, this thesis has aimed at showing that the specific formal properties of source code have a functional purpose of enabling epistemic action based on understanding of a machine language and a problem domain, itself conjugated in various contexts.

Source code, as the base of software, belongs first and foremost to the

technological realm, embodying a function and an intent of what should be achieved. Its aesthetics are thus inscribed within this technological essence by enabling the communication of the complex ideas which constitute the basis of software (its ideal version, as opposed to its implemented version, and its process of implementation, as opposed to its result).

While psychological studies and consolidated practical knowledge have shown that particular kinds of layout and presentation are beneficial to program understanding (see section 3.2.3), this is only one aspect of the system of aesthetic properties. Aesthetic values in source code are also based on the context in which said source code is written or read. These values, while varying, are nonetheless recurrently depending on a skilled relation with the program, the machine, and the audience of the program, as well as the intent of the use.

In order to achieve this epistemic function, and due to software's ambiguous nature as an *abstract artifact* (see 3.2.1), a variety of aesthetic domains are summoned by programmers in order to make sense of what they describe as occurrences of software beauty. Looking specifically at the overlap of these domains, we have shown that each aim at facilitating a transition between the surface-level syntax immediately accessible to the reader to the deep-level semantics of the topic at hand. Respectively, literature aims at evoking themes and narratives (section 4.2), architecture aims at evoking atmospheres and functions (section 4.3), while mathematics tries to communicate theorems (4.4) and engineering focuses on structural integrity and efficiency (4.4.3), with all domains above modulated by an approach to craft as a personal, hands-on skill.

From the perspective of aesthetic theory, these findings also contribute to a conception of aesthetics as a communicative endeavour. Specifically, we have shown that the concept of aesthetics amongst programmers is not seen exclusively as autotelic, but rather as a possible means to accurately represent *something* to *someone*, which falls in line with the works

of Goodman and Parsons and Carlson (Goodman, 1976; Parsons et al., 2012). As such, source code aesthetics acts as an expressive interface between a concept, a technology and two distinct individuals. Located within the particular techno-social environment of source code, this communicative role is also subject to relatively clear assessments of success or failure. A successful communication is a communication which is correctly interpreted, whereby the original ideas transmitted from the writer via the program text are found in an equivalent representation in the reader, and enable further effective action. Here, the interpretation is, at minima, what the program does, and what the program intents to do, things that might not always be aligned, resulting in the provision of agency in correctly predicting the implications of the program's operations and in the ability to correctly modify the program.

The contributions of this thesis have therefore been in the development of an aesthetic understanding of source code through an interdisciplinary analysis of a discourse analysis, drawing across media studies (from literature to software studies), science and technology studies and aesthetic philosophy. These discourses were composed of a corpus of both program-texts and commentaries and analyses by practitioners of those program texts—analyzing 65 selected source code snippets. In this sense, we have extended on the contributions of Paloqué-Bergès and MacLean and Cox by applying on their concepts of *double-meaning* and *double-coding* and showing how this co-existence of computer meaning and human meaning extends beyond the more creative writings of source code, and across communities of source code writers (see 5.2.1).

In doing so, we have also confirmed and extended Piñeiro's work on describing code aesthetics as instrumental action, bridging his field of research of software developers with other kinds of source code, and confronting it with specific example. While Piñeiro's work thoroughly explores programmers' perspectives and discourses, it does not extend its findings to other aesthetic practices mentioned by programmers—by con-

necting it to related field, we inscribe the practice of programming within a wider field of creative practices.

We proposed a conceptualization of code as semantic matter, from which executable structures are built. This approach builds on Katherine Hayles' distinction between the media properties of print and code—the former being flat, the latter being deep—and has shown the aesthetic implications of such a distinction. The contribution was to enrich our understanding of what code depth is made of, and how surface-level syntactical tokens enable the creation of deep conceptual structures.

Based on Cayley's scalar approach, we offered a typology of aesthetic properties in code, based on the purpose of aesthetics as a communicative endeavour with specific outcomes (see 5.2.2). At the level of structure, we saw that folder and file structure first provided a macro level of familiarity with the kind of application being developed, relying on convention and *family resemblance*, that data modelling offered an expressive way to communicate the important aspects of the problem domain being operated on, and that groups of statements resulted both in local coherence of operations and conceptual symmetry as these operations were repeated across the program text. At the syntax level, it is the programming language which influences the aesthetic experience of the programmer, by involving both idiomatic writing and linguistic references, as a way to demonstrate concise and precise knowledge of ones tool. Finally, the level of vocabulary enables the most expressiveness for the programmer, with both typographical parameters such as character length and casing, but also grammatical parameters, such as naming functions as verbs and variable as nouns, ultimately enabling a practice of *double-meaning*.

This complements the perspectives provided in Oram and Wilson's edited volume (Oram & Wilson, 2007). Instances of beautiful code have been given a practical framework as a way to identify positive aesthetic properties, beyond their praise by highly-skilled professionals.

Through an empirical take, we have also qualified how Florian Cramer

```

my $strength = $ARGV[0] + 1;

while(not fork){
    exit unless -$strength;
    print 0;
    twist: while(fork) {
        exit unless -$strength;
        print 1;
    }
    goto 'twist' if -$strength;
}

```

Listing 81: `forkbomb.pl` is an artwork in the exhibited sense of the term, displaying conciseness and metaphorical expression along with expressive power through its technical expansion, all the while breaking the expectation for a program not to overload the hardware on which it runs.

and his approach to source code as a form of magic relies on very concrete technical processes and habits across practices of source code writing. Building on the work of Alfred Gell describing art as the enchantment of technology, we have explicated what exactly are the complex technical hurdles and associated skills required to understand software (section 3.2.2). If there is magic in software, it is also manifested through the artistic appreciation of source code, particularly through hacking (section 2.1.2) and code poetry (section 2.1.4), and exemplified in works like `forkbomb.pl` (see listing 81).

In fact, `forkbomb.pl` is a program whose function is to replicate itself until the computer on which it runs crashes. By subverting the expectation of what a functional artefact should do, what is allowed, and what is not, this program text suggests more fundamental questions about what we expect from automatic productive processes. Nonetheless, it also gains artistic value from using the aesthetic devices of compression and double meaning, particularly in the use of `my $strength` variable name.

Finally, this thesis has contributed to a text-based approach to software aesthetics, as compared to execution-based approaches, in which source

code syntax and semantic tend to be secondary. Within those studies of code-dependent aesthetics, such as interface design (Fishwick, 2000) or creative coding (Cox & Soon, 2020), the aim was to provide an account of what code, considered as the material of software, offers in terms of representational specificities to enrich and complement those studies. Without directly contradicting any of the work mentioned in our literature review (see 1.1.3), our conclusions offer a detailed account of the material origins upon which subsequent interpretations of code are based.

6.2.1 Limitations

The first and most obvious of the limitations of this work is that a lot of source code is not accessible. While originally a freely-circulating commodity, the emergence of proprietary software at the dawn of the 1980s (see 2.1.1) has drastically limited free and open access to source code. As such, most of the source code written by software engineers in a commercial context remains confidential. For hackers, due to the nature of the work as an *ad hoc* and localized practice, few examples are made publicly available, as they are often enmeshed in more commercial projects, themselves subject to property restrictions, or in personal, *ad hoc* projects.

A second limitation is the expertise level required not just in programming, but in idiomacity—that is, in knowing how to best phrase an action in a specific language, as addressed in subsection 5.1.3—and, to a lesser extent, in the relevant problem domains. This implication of having already a solid grasp on the technical and problem context for an aesthetic judgment can have affected the accuracy of the analyses that I have given in this thesis. Consequently, it is inevitable that other experts programmers might have different opinions given their personal styles and backgrounds.

Finally, our focus on the knowledge-component of both aesthetics and source code has led us to venture into the application of cognitive sciences to fields such as programming, literature or architecture. Since this is still a

burgeoning endeavour of active research, some of the connections evoked by the current literature between code and cognition, or beauty and cognition are still bound to evolve.

6.3 Opening

Grounded in media studies and aesthetic philosophy, this thesis has nonetheless aimed at expanding the domain of what is traditionally considered beautiful, and how it is considered so, by examining the relations between beauty, function and knowledge in the specific medium of source code. Drawing on an interdisciplinary approach, the outcomes of this research therefore have some impact on both the arts and sciences in general, and programming in particular.

Deliberately eschewing notions of the artistic in favor of the beautiful, the definition work at the beginning of this thesis implicitly hypothesized that studies of beauty decoupled from art can be rich and fruitful, revealing a plethora of practices focusing on making something nice, rather than, e.g., sublime. This thesis is therefore inscribed in aesthetics of the everyday, and would suggest ways to apply aesthetic judgments to objects of study usually excluded from the aesthetic realm. Additionally, we have shown how such an object—source code—possess mechanics of meaning-making of their own, enabling unique semantic structures.

We also consider implications for programmers and craftspeople. Not that they need this work to realize that aesthetics and functionality are deeply intertwined, but rather as an explicit account of the ways in which this entanglement happens. For programmers, keeping in mind notions of scale, distance and metaphor within a particular source code would support better work. For other creators, we hope this would encourage them to investigate what is it that makes their material unique, and how it relates to other disciplines, and how formal arrangements can be rigorously

thought about, as an effective communication medium.

Ultimately, this work also has ethical implication. Knowledge, by enabling one's agency, supports and encourages good work, as opposed to meaningless labour. By organizing program texts in such a way that the next individual can discover and understand underlying concepts transmitted through the medium of source code, and then build on and complement this knowledge with their own contribution, one engages in an ethically altruistic behaviour, as opposed to self-reflexive references.

In closing, we see two main directions which can spring from this thesis, exploring the intricacies of computer-readable knowledge management, and the worldmaking of code.

The unfolding of digital media in the second half of the twentieth century has been seen as an epochal shift, along with other technologies of information reproduction and diffusion. However, computational media is specific insofar as it can be compressed and presented under various forms (from electricity to three-dimensional graphical environments and highly-dimensional vector spaces in recent machine learning approaches). How does the shape of software impact knowledge management and transmission, not just for programmers, but for end users as well, starting from those in the information sciences such as librarians, educators, journalists, researchers, and expanding to anyone engaging in a meaning-making work within a computer environment. While aesthetics can help to signify complex concepts within source code, do those concepts translate at other interface levels, or do these subsequent levels hold aesthetics principles of their own? How can the malleability of code help understanding at various levels of representation?

In terms of worldview, or how the particular structure of a text has a particular effect on an audience, the question would be to which extent

does source code structure model and affect the "real world"². Particularly in terms of time and space, as we have seen how the execution of source code engages in a deeply different scale of both of these components of our experience of reality. In terms of modelling, we could ask does a particular data structure, in how it is written, reveal social, political and economical agency? To what extent do languages such as Rust, Java or JavaScript influence the programmer's perception of the world? What is the world-view of a compiler? Could that effective impact be observed in an empirical manner? This move from static form to dynamic action would look at code's consequences beyond programmers and towards society at larg, all the while remaining grounded in a materialist approach. This relationship between form-giving and meaning-making in digital environments might start with those who write source code and compose electrical circuits, but ultimately affect all whose lives are tangled with computers.

²Throughout this work, we have been referring to the "real world" as the problem domain.

Bibliography

(2009). EARLY EXPLORATIONS: 1950S AND 1960S. In N. J. Nilsson (Ed.) *The Quest for Artificial Intelligence*, (pp. 47–48). Cambridge: Cambridge University Press.

4.4.7, J. F. (2003). Elegant. <http://www.catb.org/~esr/jargon/html/E/elegant.html>.

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.

Akten, M. (2016). A journey through multiple dimensions and transformations in SPACE.

Alexander, C. (1979). *The Timeless Way of Building*. Oxford University Press.

Alexander, C. (1996). Keynote Speech to the 1996 OOPSLA Convention.

Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., & Angel, S. (1977). *A Pattern Language: Towns, Buildings, Construction*. New York: Oxford University Press.

Allgeier, B. (2021a). Clipboard.js.

Allgeier, B. (2021b). Query.php.

Allgeier, B. (2021c). Route.php.

Allgeier, B. (2022). Kirby.

- amit (2012). Answer to "Quicksort: Iterative or Recursive".
- 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.
- Aristotle (2006). *Metaphysics*. Stilwell, KS : Digireads.com.
- Arnaud, N. (1968). *Poèmes ALGOL*. Temps mélées.
- Arns, I. (2005). Code as performative speech act. *Artnodes*, 0(4).
- Arpaci-Dusseau, R. (2023). Ostep-code.
- Arpaci-Dusseau, R. H., & Arpaci-Dusseau, A. C. (2018). *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 1.00 ed.
- Bakhtin, M. M. M. (1981). *The Dialogic Imagination : Four Essays*. Austin : University of Texas Press.
- Barker, J. (2009). Mathematical beauty. *Sztuka I Filozofia (Art and Philosophy)*, 35.
- Barrera, H. O. (2022). How the clipboard works. <https://whynothugo.nl/journal/2022/10/21/how-the-clipboard-works/>.
- Barthes, R. (1977). *S-Z*. New York: Hill & Wang.
- Barthes, R. (1984). *Le bruissement de la langue: essais critiques IV*. Paris: Seuil.
- Beardsley, M. C. (1962). The Metaphorical Twist. *Philosophy and Phenomenological Research*, 22(3), 293–307.

- Beardsley, M. C. (1970). The Aesthetic Point of View. *Metaphilosophy*, 1(1), 39–58.
- Beckett, S. (1982). *Molloy - Samuel Beckett*. Paris: Les Éditions de Minuit.
- Berry, D. M. (2011). *The Philosophy of Software: Code and Mediation in the Digital Age*. Palgrave-Macmillan.
- Berry, K. (2022). TeX-Live.
- Bertram, I. (2012). *Code {poems}*. Barcelona: Imprenta Badia.
- Biancuzzi, F., & Warden, S. (Eds.) (2009). *Masterminds Of Programming*. O'Reilly Media.
- Binkley, D., Davis, M., Lawrie, D., & Morrell, C. (2009). To camelcase or under_score. In *2009 IEEE 17th International Conference on Program Comprehension*, (pp. 158–167).
- Black, M. (1955). Metaphor. *Proceedings of the Aristotelian Society*, 55(1), 273–294.
- Black, M. J. (2002). The art of code. *Dissertations available from ProQuest*, (pp. 1–228).
- Bogost, I. (2008). The Rhetoric of Video Games. In K. Salen (Ed.) *The Ecology of Games: Connecting Youth, Games and Learning*. Cambridge, MA: The MIT Press.
- Bootz, P. (2005). The Problem of Form Transitoire Observable, A Laboratory For Emergent Programmed Art.
- Bouchardon, S. (2014). *La valeur heuristique de la littérature numérique*. Cultures numériques. Editions Hermann.
- 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.

- Box, G. E. P. (1976). Science and Statistics. *Journal of the American Statistical Association*, 71(356), 791–799.
- Bragdon, A., Reiss, S. P., Zeleznik, R., Karumuri, S., Cheung, W., Kaplan, J., Coleman, C., Adeputra, F., & LaViola, J. J. (2010). Code bubbles: Rethinking the user interface paradigm of integrated development environments. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE '10*, (pp. 455–464). New York, NY, USA: Association for Computing Machinery.
- Brand, S. (Ed.) (2019). *Code::Art::o. code::art*.
- Bray, T. (2007). Finding Things. In *Beautiful Code*. O'Reilly Media.
- 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/2019/06/11/final-a-historical-marker-that-talks-about-something-important/>.
- Brooks Jr, F. P. (1975). *The Mythical Man-month: Essays on Software Engineering*. Addison-Wesley Publishing Company.
- Brown, A., & Wilson, G. (2011). The Architecture of Open Source Applications (Volume 1)Introduction. <https://aosabook.org/en/v1/intro1.html>.
- Bryant, G. (2022). Beautiful Software. <https://www.buildingbeauty.org/beautiful-software>.
- Bush, D. (2015). 15 Ways to Write Beautiful Code - DZone DevOps. <https://dzone.com/articles/15-ways-to-write-beautiful-code>.
- Butler, B. (2012). On Programmer Archaeology.
- 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.
- Camus, A. (1972). Noces à Tipasa. In *Noces*. Paris: Gallimard.
- Cañas, J., & Antolí, A. (1998). The role of working memory in measuring mental models.
- Cant, SN., Jeffery, DR., & Henderson-Sellers, B. (1995). A conceptual model of cognitive complexity of elements of the programming process. *Information and Software Technology*, 37(7), 351–362.
- Carlson, C. (2010). The Mathematica One-Liner Competition—Wolfram Blog. <https://blog.wolfram.com/2010/12/17/the-mathematica-one-liner-competition/>.
- Carroll, N. (2002). Aesthetic Experience Revisited. *The British Journal of Aesthetics*, 42(2), 145–168.
- Cayley, J. (2012). The Code is not the Text (Unless It Is the Text) › electronic book review.
- Cellucci, C. (2015). Mathematical Beauty, Understanding, and Discovery. *Foundations of Science*, 20(4), 339–355.
- Ceruzzi, P. E. (2003). *A History of Modern Computing*. History of Computing. Cambridge, MA, USA: MIT Press, 2 ed.
- Chandra, V. (2014). *Geek Sublime: The Beauty of Code, the Code of Beauty*. Graywolf Press.
- Chatterjee, A., & Vartanian, O. (2016). Neuroscience of aesthetics. *Annals of the New York Academy of Sciences*, 1369(1), 172–194.
- Cheney, D. (2019). Practical Go: Real world advice for writing maintainable Go programs. https://dave.cheney.net/practical-go/presentations/gophercon-singapore-2019.html#_guiding_principles.

- 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.
- Cities, H. (2022). Peter van Hardenberg - Why Can't We Make Simple Software?
- 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, E. G. (2012). *Coding Freedom: The Ethics and Aesthetics of Hacking*. Princeton: Princeton University Press.
- 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.
- Confreaks, & Hickey, R. (2012). Rails Conf 2012 Keynote: Simplicity Matters by Rich Hickey.
- Cox, G. (2011). The Aesthetics of Generative Code.
- Cox, G., & McLean, C. A. (2013). *Speaking Code: Coding as Aesthetic and Political Expression*. MIT Press.

- Cox, G., & Soon, W. (2020). *Aesthetic Programming: A Handbook of Software Studies*. Open Humanities Press.
- CPPP Conference (2022). Keynote: On the Aesthetics of Code - Sy Brand - CPPP 2021.
- Cramer, F. (2003). *Words Made Flesh*. Piet Zwart Institute.
- Craver, S. (2015). The Underhanded C Contest » About.
- Croll, A. (2014). *If Hemingway Wrote JavaScript*. No Starch Press.
- 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*.
- Depaz, P. (2021). The Craft of Code: Practice and Knowledge in the Production of Software. *Kuckuck*, 1.
- Depaz, P. (2022). Discursive Strategies in Style Guides Negotiation on GitHub. *RESET. Recherches en sciences sociales sur Internet*, (11).
- Depaz, P. (2023). Stylistique de la recherche linguistique en IA: De LISP à GPT-3. In A. Gefen (Ed.) *Créativités Artificielles – La Littérature et l'art à l'heure de l'intelligence Artificielle*. Les Presses du Réel.
- Detienne, F. (2001). *Software Design – Cognitive Aspect*. Springer Science & Business Media.
- Deutsche Bundestag (2022). GG - Grundgesetz für die Bundesrepublik Deutschland. <https://www.gesetze-im-internet.de/gg/BJNRoooo10949.html>.

- Dexter, S., Dolese, M., Seidel, A., & Kozbelt, A. (2011). On the Embodied Aesthetics of Code. *Culture Machine*, 12(1).
- Dicola, T. (2015). Pi_video_looper/video_looper.py at master · adafruit/pi_video_looper · GitHub. <https://archive.softwareheritage.org/swh:1:cnt:cd50aeb12dof1doa3476ao667a584oa64269dbb6;origin=https://github.com/adafruit/pi-video-looper>
- 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. (1965). Cooperating sequential processes (EWD-123). Tech. Rep. EWD-123, Eindhoven Technological University.
- Dijkstra, E. W. (1968). Letters to the editor: Go to statement considered harmful. *Communications of the ACM*, 11(3), 147–148.
- Dijkstra, E. W. (1972). Chapter I: Notes on structured programming. In *Structured Programming*, (pp. 1–82). Academic Press Ltd.
- Dijkstra, E. W. (1975). How do we tell truths that might hurt?
- 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.
- Downton, P. (1998). On Knowledge In Architecture and Science.
- du Gay, P., Hall, S., Janes, L., Madsen, A. K., Mackay, H., & Negus, K. (2013). *Doing Cultural Studies: The Story of the Sony Walkman*. Los Angeles, CA: SAGE Publications Ltd, second edition ed.
- Du Maurier, D. (1938). *Rebecca*. Garden City, NY: Doubleday & Company, [book club edition] ed.
- Duff, T. (1983). Tom Duff on Duff's Device.
<http://www.lysator.liu.se/c/duffs-device.html>.
- Efatmaneshnik, M., & Ryan, M. J. (2019). On the Definitions of Sufficiency and Elegance in Systems Design. *IEEE Systems Journal*, 13(3), 2077–2088.
- Elgin, C. Z. (1993). Understanding: Art and science. *Synthese*, 95(1), 13–28.
- Elgin, C. Z. (2011). Making Manifest: The Role of Exemplification in the Sciences and the Arts. *Principia: An International Journal of Epistemology*, 15(3), 399–413.
- Elgin, C. Z. (2017). *True Enough*. Boston: The MIT Press.
- Elgin, C. Z. (2020). Understanding Understanding Art. In V. Granata, R. Pouivet, & Université de Lorraine (Eds.) *Épistémologie de l'esthétique: perspectives et débats*, Collection "Hors-série". Rennes: Presses universitaires de Rennes.
- Ensmenger, N. L. (2012). *The Computer Boys Take Over: Computers, Programmers, and the Politics of Technical Expertise*. Cambridge, Mass.: The MIT Press.
- Ershov, A. P. (1972). Aesthetics and the human factor in programming. *Communications of the ACM*, 15(7), 501–505.

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. (2000). Aesthetic Programming.

Fishwick, P. A. (Ed.) (2006). *Aesthetic Computing*. The MIT Press.

FLUSSER, VILÉM., & 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. (2009). TwoHardThings. <https://martinfowler.com/bliki/TwoHardThings.html>.

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.

Fuller, U., Johnson, C. G., Ahoniemi, T., Cukierman, D., Hernán-Losada, I., Jackova, J., Lahtinen, E., Lewis, T. L., Thompson, D. M., Riedesel, C., &

- Thompson, E. (2007). Developing a computer science-specific learning taxonomy. *ACM SIGCSE Bulletin*, 39(4), 152–170.
- Gabriel, R. P. (1998). *Patterns of Software: Tales from the Software Community*. Oxford University Press.
- Galloway, A. R. (2006). Language Wants To Be Overlooked: On Software and Ideology. *Journal of Visual Culture*, 5(3), 315–331.
- Galloway, A. R. (2012). *The Interface Effect*. Cambridge, UK ; Malden, MA: Polity, 1st edition ed.
- Gamma, E., Helm, R., Johnson, R., Vlissides, J., & Booch, G. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Mass: Addison-Wesley Professional, 1st edition ed.
- Gannon, J. D., & Horning, J. J. (1975). The impact of language design on the production of reliable software. *ACM SIGPLAN Notices*, 10(6), 10–22.
- Garfinkel, S. (2000). Biological Computing. <https://www.technologyreview.com/2000/05/01/236304/biological-computing/>.
- Garland, D. (2000). Software Architecture: A Roadmap. In *The Future of Software Engineering*. ACM Press, anthony finkelstein (ed.) ed.
- Garry, C., & Hamilton, M. (1969). LU-NAR_LANDING_GUIDANCE_EQUATIONS.agc. <https://archive.softwareheritage.org/swh:1:cnt:b269a39d9d92fa5e443344e03403966daf503eb5;origin=ht11;visit=swh:1:snp:3124c2317c6f820c9f84343a220790c8810f63df;anchor=swh:1:rev:b56b8c3d03e810a6ceb6670>.
- 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.
- Gentner, D., & Nielsen, J. (1996). The Anti-Mac interface. *Communications of the ACM*, 39(8), 70–82.
- 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.
- Goody, J. (1986). *The Logic of Writing and the Organization of Society*. Studies in Literacy, the Family, Culture and the State. Cambridge University Press.
- Goriunova, O., & Shulgin, A. (2005). *Read Me: Software Art & Cultures*. Aarhus: Aarhus University Press, 2004th edition ed.
- Graham, G. (2000). Architecture as an Art. In *Philosophy of the Arts*. Routledge, 2 ed.
- Graham, P. (2003). Hackers and Painters.
<http://www.paulgraham.com/hp.html>.

- Granger, G.-G. (1988). *Essai d'une philosophie du style*. Paris: Odile Jacob / Seuil, édition revue et corrigée ed.
- Green, R. (2006). How To Write Unmaintainable Code.
<https://archive.ph/Pn5hH>.
- Greenberg, B. S. (1996). Multics Emacs History/Design/Implementation.
<https://www.multicians.org/mepap.html#secviii>.
- Grudin, J. (2016). From Tool to Partner: The Evolution of Human-Computer Interaction. *Synthesis Lectures on Human-Centered Informatics*, 10(1), i-183.
- Guerra, E., & Manns, M. L. (2022). PLoP 2022 - 29th Conference on Pattern Languages of Programs (online).
<https://hillside.net/plop/2022/index.php?nav=ploppaperscfp>.
- 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.
- Hardy, G. H. (2012). *A Mathematician's Apology*. Canto Classics. Cambridge: Cambridge University Press.
- 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.
- Heidegger, M., & Hofstadter, A. (1975). Building, Dwelling, Thinking. In *Poetry, Language, Thought*, Harper Colophon Books. New York: Harper & Row, 1st harper colophone ed ed.
- Heinonen, A., Lehtelä, B., Hellas, A., & Fagerholm, F. (2023). Synthesizing research on programmers' mental models of programs, tasks and concepts — A systematic literature review. *Information and Software Technology*, (p. 107300).
- 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.
- Hermans, F., Chahchouhi, M., & Al-Ers, H. (2020). The Role of Aesthetics in Code: A Qualitative Interview Study with Professionals. In *PX/20*. Porto, Portugal.
- 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.
- Hoare, C. A. R. (1981). The emperor's old clothes. *Commun. ACM*, 24(2), 75–83.
- Hoare, C. A. R. (1996). How did software get so reliable without proof? In M.-C. Gaudel, & J. Woodcock (Eds.) *FME'96: Industrial Benefit and Advances in Formal Methods*, Lecture Notes in Computer Science, (pp. 1–17). Berlin, Heidelberg: Springer.
- Holden, M. D., & Kerr, M. C. (2016). *./Code –Poetry*. S.L: CreateSpace Independent Publishing Platform, 1.0 edition ed.
- Hopkins, S. (1992). Camels and Needles: Computer Poetry Meets the Perl Programming Language. *Usenix Technical Conference*.
- Houkes, W., & Vermaas, P. (2004). Actions Versus Functions: A Plea for an Alternative Metaphysics of Artifacts*. *The Monist*, 87(1), 52–71.
- 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.
- Ingalls, D., Kaehler, T., Maloney, J., Wallace, S., & Kay, A. (1997). Back to the future: The story of Squeak, a practical Smalltalk written in itself. *ACM SIGPLAN Notices*, 32(10), 318–326.
- Inglis, M., & Aberdein, A. (2015). Beauty is Not Simplicity: An Analysis of Mathematicians' Proof Appraisals. *Philosophia Mathematica*, 23(1), 87–109.
- interstar (2017). Quality Without A Name (QWAN) examples?

Investigator, Q. (2018). In Theory There Is No Difference Between Theory and Practice, While In Practice There Is – Quote Investigator.

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.

Jackson, K. (2010). "Perfection is achieved, not when there is nothing more to add, but when there i... | Hacker News. <https://news.ycombinator.com/item?id=1640594>.

Jacques, J. T., & Kristensson, P. O. (2015). Understanding the effects of code presentation. In *Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools*, PLATEAU 2015, (pp. 27–30). New York, NY, USA: Association for Computing Machinery.

James, G. (1987). *The Tao of Programming*. InfoBooks.

Jameson, F. (1991). *Postmodernism Or, The Cultural Logic Of Late Capitalism*. UK: Verso.

Jeantet, A. (1998). Les objets intermédiaires dans la conception. Éléments pour une sociologie des processus de conception. *Sociologie du travail*, 40(3), 291–316.

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.

- Jullien, C. (2012). From the Languages of Art to Mathematical Languages, and back again. *Enrahonar, Quaderns de Filosofia*, 49, 91–105.
- Kagen, M., & Werner, K. J. (2016). Code Poetry Slam. <https://web.archive.org/web/20161024152353/http://stanford.edu/~mkagen/codepoetryslam/>.
- Kanakarakis, I. (2022). 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).
- Kay, A. C. (1993). The early history of Smalltalk. *ACM SIGPLAN Notices*, 28(3), 69–95.
- Keith, J. (2016). Resilient Web Design - Chapter 2. <https://resilientwebdesign.com/chapter2/#The%20world%20of%20architecture%20has%20accrued%20a%20lot%20of%20experience>
- Keller, A. (2021). Des textes d'algorithmes concis, quelques exemples tirés de sūtras d'Āryabhatā 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. (1981). Why Pascal is Not My Favorite Programming Language. <https://www.lysator.liu.se/c/bwk-on-pascal.html>.
- Kernighan, B. W. (2007). A Regular Expression Matcher. In *Beautiful Code*. O'Reilly Media.
- Kernighan, B. W., & Plauger, P. J. (1978). *The Elements of Programming Style*, 2nd Edition. New York: McGraw-Hill, 2nd edition ed.

- Kidd, E. (2005). Why Ruby is an acceptable LISP (2005) | Random Hacks. <http://www.randomhacks.net/2005/12/03/why-ruby-is-an-acceptable-lisp/>.
- Kintsch, W., & van Dijk, T. A. (1978). Toward a model of text comprehension and production. *Psychological Review*, 85(5), 363–394.
- Kirchner, M. (2022a). Linked List. <https://archive.softwareheritage.org/swh:1:cnt:78b25703fda5206fd4c9ec> list-good-taste;visit=swh:1:snp:5161a108062af8d87cb24b4568dc8a1b8oed197;anchor=swh:1:rev:cfc3fb918.
- Kirchner, M. (2022b). Linked List - Removing. <https://archive.softwareheritage.org/swh:1:cnt:6dd41adbff62aa9cd5a310690d5b3943ae52c1bf>;origin=https://archive.softwareheritage.org/swh:1:cnt:6dd41adbff62aa9cd5a310690d5b3943ae52c1bf;visit=swh:1:snp:5161a108062af8d87cb24b4568dc8a1b8oed197;anchor=swh:1:rev:cfc3fb942.
- 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.
- Kline, R. B., & Seffah, A. (2005). Evaluation of integrated software development environments: Challenges and results from three empirical studies. *International Journal of Human-Computer Studies*, 63(6), 607–627.
- Knuth, D. (2011). *The Art of Computer Programming: Combinatorial Algorithms, Volume 4A, Part 1*. Upper Saddle River, NJ: Addison-Wesley Professional, 1st edition 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.
- Koenemann, J., & Robertson, S. P. (1991). Expert problem solving strategies for program comprehension. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '91, (pp. 125–130). New York, NY, USA: Association for Computing Machinery.
- Korte, T. (2010). Frege's Begriffsschrift as a lingua characteristica. *Synthese*, 174(2), 283–294.
- Kozbelt, A., Dexter, S., Dolese, M., & Seidel, A. (2012). The aesthetics of software code: A quantitative exploration. *Psychology of Aesthetics, Creativity, and the Arts*, 6(1), 57–65.
- Kragh, H. (2002). Paul Dirac: Seeking beauty. *Physics World*, 15(8), 27.
- Kristensen, B. B. (1994). Complex associations: Abstractions in object-oriented modeling. *ACM SIGPLAN Notices*, 29(10), 272–286.
- Kulkarni, N., & Varma, V. (2017). Supporting comprehension of unfamiliar programs by modeling cues. *Software Quality Journal*, 25(1), 307–340.
- Kurp, P. (2008). Green computing. *Communications of the ACM*, 51(10), 11–13.
- Lakoff, G. (1980). Conceptual Metaphor in Everyday Language. *The Journal of Philosophy*, 77(8).
- Lakoff, G., & Johnson, M. (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).
- Langston, J. (2017). A_mind_is_born.asm.
https://archive.softwareheritage.org/browse/content/sha1_git:4990e2fef33de535851ccc6f6076e1d7ef6c
- Latour, B. (2007). *Reassembling the Social: An Introduction to Actor-Network-Theory*. Clarendon Lectures in Management Studies. Oxford, New York: Oxford University Press.
- Laurel, B. (1993). *Computers as Theatre*. Addison-Wesley.
- Lavocat, F. (Ed.) (2015). *Interprétation littéraire et sciences cognitives*. Paris: Hermann.
- Le Corbusier, & Saugnier (1923). *Vers une architecture*. Paris: G. Crès.
- Le Lionnais, F. (1971). *Great Currents of Mathematical Thought*. New York, Dover Publications.
- Leddy, T., & Puolakka, K. (2021). Dewey's Aesthetics. In E. N. Zalta (Ed.) *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, fall 2021 ed.
- Leder, H., Belke, B., Oeberst, A., & Augustin, D. (2004). A model of aesthetic appreciation and aesthetic judgments. *British Journal of Psychology*, 95(4), 489–508.
- Ledgard, R. G., Henry (2011). Coding Guidelines: Finding the Art in the Science. <https://cacm.acm.org/magazines/2011/12/142527-coding-guidelines-finding-the-art-in-the-science/fulltext>.
- Lee, E. A. (2006). The Problem with Threads. Tech. Rep. UCB/EECS-2006-1, EECS Department, University of California, Berkeley.

- Leighton, J. A. (1907). The Objects of Knowledge. *The Philosophical Review*, 16(6), 577–587.
- Leppäjärvi, J. (2008). A pragmatic, historically oriented survey on the universality of synchronization primitives. Tech. rep., University of Oulu.
- 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.
- 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.
- Linsky, B., & Irvine, A. D. (2022). Principia Mathematica. In E. N. Zalta (Ed.) *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, spring 2022 ed.
- Linux (2023). Fs/ext4/resize.c.
- Linux Information Project (2006). Source code definition by The Linux Information Project. <http://linfo.org/sourcecode.html>.
- Lions, J. (1996). *Lions' Commentary on UNIX 6th Edition with Source Code. Peer-to-Peer Communications*.

- Lynch, K. (1959). *The Image Of The City*. MIT Press.
- Macé, M. (2016). *Styles: Critique de Nos Formes de Vie*. Gallimard, nrf essais ed.
- Mackenzie, A. (2006). *Cutting Code: Software and Sociality*. Peter Lang.
- 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*.
- Matsumoto, Y. (2007). Treating Code as an Essay. In A. Oram, & G. Wilson (Eds.) *Beautiful Code: Leading Programmers Explain How They Think*. Beijing ; Sebastopol, Calif: O'Reilly Media, 1st edition ed.
- Matsumoto, Y. (2019). Yukihiko Matsumoto: "Ruby is designed for humans, not machines". <https://evrone.com/blog/yukihiko-matsumoto-interview>.
- 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., 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.
- McKeithen, K. B., Reitman, J. S., Rueter, H. H., & Hirtle, S. C. (1981). Knowledge organization and skill differences in computer programmers. *Cognitive Psychology*, 13(3), 307–325.
- 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.
- Mélès, B. (2017). Time and activity in Unix. *Reseaux*, 206(6), 125–153.
- Melik, D. (2012). PC demos FAQ.
- Mentor++, T. (1986). The Conscience of a Hacker.
- 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.
- Milner, R. (1996). Semantic ideas in computing. In I. Wand, & R. Milner (Eds.) *Computing Tomorrow*, (pp. 246–283). United States: Cambridge 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.
- Moore, R. (2011). Junya Ishigami: Architecture of Air; Serpentine Gallery Pavilion 2011 – review. *The Observer*.
- Moss, R. (2021a). BubbleSort.jl.
- Moss, R. (2021b). K_NearestNeighbors.jl.
- Moss, R. (2022). BeautifulAlgorithms.jl. <https://github.com/mosssr/BeautifulAlgorithms.jl>.
- Mosteirin, R. J., & Dobson, J. E. (2019). *Moonbit*. Punctum Books.
- Mullet, D. R. (2018). A General Critical Discourse Analysis Framework for Educational Research. *Journal of Advanced Academics*, 29(2), 116–142.
- Mumford, L. (1934). *Technics And Civilization*.
- Munroe, R. (2011). Good Code. <https://xkcd.com/844/>.

- Muon Ray (1985). Computer Science Lecture - Hardware, Software and Heuristics.
- Muratori, C. (2014). Semantic Compression. https://caseymuratori.com/blog_0015.
- Murray, J. H. (1998). *Hamlet on the Holodeck: The Future of Narrative in Cyberspace*. Cambridge, MA, USA: MIT Press.
- Mustacchi, R. (2019). Mac_sched.c. Joyent.
- Nabokov, V. (1980). *Lectures on Literature*. Harcourt Brace Jovanovich.
- Nather, E. (1983). The Story of Mel. <https://users.cs.utah.edu/~elb/folklore/mel.html>.
- Naur, P. (1985). Programming as theory building. *Microprocessing and Microprogramming*, 15(5), 253–261.
- Neumann, P. G. (1985). COMPUTER-RELATED INCIDENTS ILLUSTRATING RISKS TO THE PUBLIC. *The RISKS Digest, Volume 1 Issue 01*, 1(01).
- Neumann, P. G. (1990). Beauty and the Beast of Software Complexity—Elegance versus Elephants. In W. H. J. Feijen, A. J. M. van Gasteren, D. Gries, & J. Misra (Eds.) *Beauty Is Our Business: A Birthday Salute to Edsger W. Dijkstra*, Texts and Monographs in Computer Science, (pp. 346–351). New York, NY: Springer.
- 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.
- Norman, D. (2013). *The Design Of Everyday Things*. New York, New York: Basic Books, revised edition ed.
- Norvig, P. (1998). Design Patterns in Dynamic Languages.
- Oberkampf, W. L., & Roy, C. J. (2010). *Verification and Validation in Scientific Computing*. Cambridge University Press.
- Oliveira, D., Bruno, R., Madeiral, F., & Filho, F. C. (2020). Evaluating Code Readability and Legibility: An Examination of Human-centric Studies. *undefined*.
- 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. (1990a). A taxonomy for programming style. In *Proceedings of the 1990 ACM Annual Conference on Cooperation*, CSC '90, (pp. 244–250). New York, NY, USA: Association for Computing Machinery.
- Oman, P. W., & Cook, C. R. (1990b). Typographic style is more than cosmetic. *Communications of the ACM*, 33(5), 506–520.
- 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, 3 ed.
- Oram, A., & Wilson, G. (Eds.) (2007). *Beautiful Code: Leading Programmers Explain How They Think*. Beijing ; Sebastopol, Calif: O'Reilly Media, 1st edition ed.
- Ortega, M. (2011). Maca/Ruby-code-poem.
- 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.
- Overflow, S. (2013). How can you explain "beautiful code" to a non-programmer?
- Overflow, S. (2021). Stack Overflow Developer Survey 2021.
- Paloque-Bergès, C. (2009). *Poétique des codes sur le réseau informatique*. Archives contemporaines.
- Papert, S. A. (1978). The Mathematical Unconscious. In J. Wechsler (Ed.) *On Aesthetics in Science*. Cambridge, Mass.: MIT Press.
- Parsons, G., Carlson, A., Parsons, G., & Carlson, A. (2012). *Functional Beauty*. Oxford, New York: Oxford University Press.
- Parsons, J., & Wand, Y. (1997). Choosing classes in conceptual modeling. *Communications of the ACM*, 40(6), 63–69.
- Pattis, R. E. (1988). Textbook errors in binary searching. In *SIGCSE '88*.
- Peacocke, A. (2023). Aesthetic Experience. In E. N. Zalta, & U. Nodelman (Eds.) *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, spring 2023 ed.

- Pellet-Mary, A. (2020). Co6GC: Program Obfuscation | COSIC.
- Penny, S. (2019). *Making Sense: Cognition, Computing, Art and Embodiment*. MIT Press.
- Perlin, K. (2022). Chalktalk. New York University.
- Perlis, A. J. (1982). Special Feature: Epigrams on programming. *ACM SIGPLAN Notices*, 17(9), 7–13.
- Perrin, C. (2006). ITLOG Import: Elegance. <https://web.archive.org/web/20200730232944/http://sob.apotheon.org/?p=113>.
- Peters, T. (1999). Code Style. <https://docs.python-guide.org/writing/style>.
- Peters, T. (2004). PEP 20 – The Zen of Python | peps.python.org. <https://peps.python.org/pep-0020/>.
- 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.
- Polanyi, M., & Sen, A. (2009). *The Tacit Dimension*. Chicago ; London: University of Chicago Press, revised ed. edition ed.

- 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).
- Pratt, T. W., & Zelkowitz, M. V. (2000). *Programming Languages: Design and Implementation*. Upper Saddle River, NJ: Pearson, 4th edition ed.
- Programming Wisdom [@codewisdom] (2021). "A designer knows [one] has achieved perfection not when there is nothing left to add, but when there is nothing left to take away." – Antoine de Saint-Exupéry.
- 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.
- Reber, R., Schwarz, N., & Winkielman, P. (2004). Processing Fluency and Aesthetic Pleasure: Is Beauty in the Perceiver's Processing Experience? *Personality and Social Psychology Review*, 8(4), 364–382.
- 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?; Ti- etokonedemot - Mikä Saa Ne Hyrräämään?*. G3 Lisensiaatintyö, Aalto-yliopisto; Aalto University.
- Richards, I. A. (1930). *Practical Criticism*. Kegan Paul Trench Trubner And Company Limited.
- Ricoeur, P. (2003). *The Rule of Metaphor: The Creation of Meaning in Language*. Psychology Press.
- Risam, R. (2015). The poetry of executable code | Jacket2. <https://jacket2.org/commentary/poetry-executable-code>.
- Rokeyby, D. (2003). The Computer as a Prosthetic Organ of Philosophy. <http://www.dichtung-digital.de/2003/issue/3/Rokeby.htm>.
- Root-Bernstein, R. S. (2002). Aesthetic cognition. *International Studies in the Philosophy of Science*, 16(1), 61–77.
- 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.

- Russell, B. (1950). Logical positivism. *Revue Internationale de Philosophie*, 4(11), 3–19.
- Ryan, M.-L. (2009). Space. In *Space*, (pp. 420–433). De Gruyter.
- Ryan, M.-L. (2021). Four Types of Textual Space and their Manifestations in Digital Narrative. In D. Punday (Ed.) *Digital Narrative Spaces: An Interdisciplinary Examination*. Routledge.
- Ryle, G. (1951). *The Concept Of Mind*. Hutchinsons University Library.
- Sack, W. (2019). *The Software Arts*. The MIT Press.
- Saint-Exupery, A. D. (1990). *Wind, Sand and Stars*. The Folio Society, first thus edition ed.
- Saito, Y. (2012). Everyday Aesthetics and Artification. *Contemporary Aesthetics (Journal Archive)*, (4).
- Sanchez, A. (2016). Solutions of xchg rax,rax. <https://alexaltea.github.io/blog/posts/2016-10-12-xchg-rax-rax-solutions/>.
- Sanglard, F. (2018). *Game Engine Black Book: DOOM*. Software Wizards.
- 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.
- Schaeffer, J.-M. (1999). *Pourquoi La Fiction?*. Paris: Éditions du Seuil.
- Schiffrin, D. (1994). *Approaches to Discourse*. Oxford, UK ; Cambridge, Mass., USA : B. Blackwell.
- Schmitz, B. (2015). What makes some code "beautiful"? <https://qr.ae/pyIE7R>.

- 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. (2013). *The Aesthetics of Architecture*. Princeton: Princeton University Press.
- Sebesta, R. W. (2018). *Concepts of Programming Languages*. NY, NY: Pearson, 12th edition ed.
- Secret Labs AB (2001). _parser.py. Python.
- 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.
- Seibel, P. (2014). Code is not literature. <https://gigamonkeys.com/code-reading/>.
- Sennett, R. (2009). *The Craftsman*. Yale University Press.
- Sethi, R. (1996). *Programming Languages: Concepts and Constructs*. Reading, Mass.
- Shannon, C. E. (2001). A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, 5(1), 3–55.
- Shaw, M., & Garlan, D. (1996). *Software Architecture: Perspectives on an Emerging Discipline*. Upper Saddle River, NJ: Pearson.
- Sheil, B. A. (1981). The Psychological Study of Programming. *ACM Computing Surveys*, 13(1), 101–120.

- Shiner, L. (2009). Functional beauty : The metaphysics of beauty and specific functions in architecture. *Szutka i Filozofia*, 35(78-99), 23.
- Shneiderman, B. (1977). Measuring computer program quality and comprehension. *International Journal of Man-Machine Studies*, 9(4), 465–478.
- Simmel, G. (1991). The Problem of Style. *Theory, Culture & Society*, 8(3), 63–71.
- 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.
- Sinclair, N. (2004). The Roles of the Aesthetic in Mathematical Inquiry. *Mathematical Thinking and Learning*, 6(3), 261–284.
- Sinclair, N. (2011). Aesthetic Considerations in Mathematics. *Journal of Humanistic Mathematics*, 1(1), 2–32.
- Sinclair, N., & Pimm, D. (2010). The Many and the Few: Mathematics, Democracy and the Aesthetic.
- Smith, B. C. (1998). *On the Origin of Objects*. Cambridge, Mass.: A Bradford Book, reprint edition ed.
- Smith, B. C. (2016). AoS V1·Co: Introduction. <https://web.archive.org/web/20160826234606/http://ageofsignificance.org/aos/en/aos-v1co.html>.
- Soloway, E., & Ehrlich, K. (1984). Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering*, SE-10(5), 595–609.
- Soloway, E., Ehrlich, K., & Bonar, J. (1982). Tapping into tacit programming knowledge. In *Proceedings of the 1982 Conference on Human Factors in Computing Systems*, (pp. 52–57).

- Sommerville, I. (2010). *Software Engineering*. Boston: Pearson, 9th edition ed.
- Sondheim, A. (2001). Introduction: Codework. *American Book Review*, 22(6).
- Spencer, H. (1994). The Ten Commandments for C Programmers (Annotated Edition). <https://www.lysator.liu.se/c/ten-commandments.html>.
- Spinellis, D., & Gousios, G. (2009). *Beautiful Architecture: Leading Thinkers Reveal the Hidden Beauty in Software Design*. "O'Reilly Media, Inc.".
- 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.
- Stansifer, R. (1994). *Study of Programming Languages, The*. Englewood Cliffs, N.J: Prentice Hall, 1st edition ed.
- Starikova, I. (2018). Aesthetic Preferences in Mathematics: A Case Study†. *Philosophia Mathematica*, 26(2), 161–183.
- 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.
- Suber, P. (1988). What is Software? *Journal of Speculative Philosophy*, 2(2), 89–119.
- Sullivan, L. H. (1896). *The Tall Office Building Artistically Considered*.
- Sustrik, M. (2021). On the Nature of Programming Languages. <https://250bpm.com/blog:152/index.html>.

- 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. (2015). Daniel Temkin | Folders Language. <http://danieltemkin.com/Esolangs/Folders/>.
- Temkin, D. (2017). Sentences on Code Art. <https://esoteric.codes/blog/sentences-on-code-art>.
- Thomas, R. (2017). Beauty is not all there is to Aesthetics in Mathematics†. *Philosophia Mathematica*, 25(1), 116–127.
- Thompson, D. V. (1934). The Study of Medieval Craftsmanship. *Bulletin of the Fogg Art Museum*, 3, 3–8.
- Tirrell, J. (2012). Dumb People, Smart Objects: The Sims and The Distributed Self. In *The Philosophy of Computer Games Conference*.
- Tomov, L. (2016). The Role of Aesthetics in Software Design, Development and Education: Review and Definitions. *Computer Science and Education in Computer Science*, 12(1), 1–16.
- Torvalds, L. (2016). Linus Torvalds: The mind behind Linux | TED Talk. https://www.ted.com/talks/linus_torvalds_the_mind_behind_linux.
- 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).

Tufte, E. R. (2001). *The Visual Display of Quantitative Information*. Graphics Press.

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.

Unknown (2017). Source Code Poetry | About.
<https://www.sourcecodepoetry.com>.

Van Roy, P. (2012). Programming Paradigms for Dummies: What Every Programmer Should Know.

Vardi, M. Y. (2010). Science has only two legs. *Communications of the ACM*, 53(9), 5.

Vicious, K. (2008). Beautiful Code Exists, if You Know Where to Look - ACM Queue. <https://queue.acm.org/detail.cfm?id=1454458>.

Victor, B. (2011a). Explorable Explanations.
<http://worrydream.com/ExplorableExplanations/>.

Victor, B. (2011b). Tangle: A JavaScript library for reactive documents.

Victor, B. (2014). Humane representation of thought: A trail map for the 21st century. In *Proceedings of the Companion Publication of the 2014 ACM*

SIGPLAN Conference on Systems, Programming, and Applications: Software for Humanity, SPLASH '14, (p. 5). New York, NY, USA: Association for Computing Machinery.

Voloshinov, V. N., & Bachtin, M. M. (1986). *Marxism and the Philosophy of Language*. Harvard University Press.

Von Neumann, J. (1947). The Mathematician. In M. J. Adler, & R. B. Heywood (Eds.) *The Works of the Mind*. Chicago: University of Chicago Press.

Waldron, R. (2020). [Idiomatic.js/readme.md at master · rwaldron/idiomatic.js](#).

Wallen, L. A. (1990). On Form, Formalism and Equivalence. In W. H. J. Feijen, A. J. M. van Gasteren, D. Gries, & J. Misra (Eds.) *Beauty Is Our Business: A Birthday Salute to Edsger W. Dijkstra*, Texts and Monographs in Computer Science, (pp. 417–426). New York, NY: Springer.

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.

WeidiDeng (2023). Caddyserver/caddy. Caddy.

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.

- Wells, D. (1990). Are these the most beautiful? *The Mathematical Intelligencer*, 12(3), 37–41.
- West, J. (1987). *Macintosh Programmers Workshop*. The Macintosh Performance Library. Bantam Books.
- Wettel, R. (2008). CodeCity.
- Wiedenbeck, S. (1991). The initial stage of program comprehension. 35(4), 517–540.
- Wikipedia (2021). Linux kernel.
- Wikipedia (2023a). Semaphore (programming). *Wikipedia*.
- Wikipedia (2023b). Unified Modeling Language. *Wikipedia*.
- Wilken, R. (2010). The card index as creativity machine. *Culture Machine*.
- 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.
- Winograd, T. (1982). *Language As a Cognitive Process: Syntax*. Reading, Mass: Addison-Wesley.
- Winograd, T., & Flores, F. (1986). *Understanding Computers and Cognition: A New Foundation for Design*. Intellect Books.
- Wirth, N. (1976). *Algorithms + Data Structures*. Prentice-Hall.
- Wirth, N. (1990). Drawing lines, circles, and ellipses in a raster. In *Beauty Is Our Business: A Birthday Salute to Edsger W. Dijkstra*, (pp. 427–434). Berlin, Heidelberg: Springer-Verlag.
- Wirth, N. (2003). The Essence of Programming Languages. In L. Böszörményi, & P. Schojer (Eds.) *Modular Programming Languages*, Lecture Notes in Computer Science, (pp. 1–11). Berlin, Heidelberg: Springer.

- 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*.
- Wordpress (2023). WordPress/WordPress. WordPress.
- Wyatt, S. (2004). Danger! Metaphors at Work in Economics, Geophysiology, and the Internet. *Science, Technology, & Human Values*, 29(2), 242–261.
- xorp^d (2014). *Xchg Rax,Rax*. Createspace independent publishing platform ed.
- 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>.
- Zenil, H. (2021). Compression is Comprehension, and the Unreasonable Effectiveness of Digital Computation in the Natural World.

List of Listings

1	Example of the basic elements of a computer program, written in Python	8
2	A very verbose way to left pad a digit with zeroes in the C language.	20
3	A very terse way to left pad a digit with zeroes in the C language.	20
4	Unicode string initialization in Microsoft 2000 operating system, with a first part showing an explicit repeating pattern, while the second part shows a more compressed approach.	58
5	Overlapping programming voices can be hinted at by different comment styles.	60
6	pnpenum.c shows the explicit traces of multiple authors collaborating on a single file over time.	61
7	The setting of whether a query should be distinct includes some verbose details which prove to be helpful in the long run (Allgeier, 2021b).	63
8	The inclusion of comments help guide a programmer through an open-source project (Allgeier, 2021c).	64
9	Even in a productive and efficient open-source project, one can detect traces of "hacks" (Allgeier, 2021a).	66
10	This program text selects all the lines from an input file which is longer than 6 characters in the C programming language. See the one-line alternative implementation in 11.	71

11	This program text selects all the lines from an input file which is longer than 6 characters in the C programming language, in just one line of code. See the alternative implementation in 20 lines of code in 10.	72
12	Conway's Game of Life implemented in APL is a remarkable example of conciseness, at the expense of readability.	73
13	This particular implementation of a function calculating the inverse square root of a number has become known in programming circles for both its speed and unscrutability.	74
14	westley.c, entry to the 1988 IOCCC	76
15	Mesh.m	83
16	The Prolog programming language focuses first and foremost on logic predicates in order to perform computation, rather than more practical system calls.	89
17	Scheme interpreter written in Scheme, revealing the power and self-reference of the language.	90
18	Bubble Sort implementation in Julia uses the language features to use only a single iteration loop. (Moss, 2021a)	93
19	Nearest neighbor implementation in Julia (Moss, 2021b).	93
20	AGC source code for the Lunar Landing Guidance Equation, 1969	99
21	Just Another Perl Hacker, japh.pl	101
22	Black Perl is one of the first Perl poems, shared anonymously online. It makes creative use of Perl's flexible and high-level syntax.	103
23	#SongsInCode is an example of functional source code poetry written to represent the traditionally non-functional domain of pop songs.	106
24	water.c has a very deliberate layout and syntax, reminiscing of <i>calligrammes</i> (Holden & Kerr, 2016).	107

25	The output of 24 consists in ASCII representation of water droplets, bearing a family resemblance to BASIC one liners, and suggesting a complementary representation of water. . . .	107
26	home.js is an excerpt of a JavaScript program text as it is written by a human programmer, before minification.	116
27	home.js is the same program as in 26, after minification. Syntactical density is gained at the expense of clarity.	117
28	Example of clarity differences between two methods.	120
29	unique.py: A function to check for the uniqueness of array elements, using a very specific feature of the Python syntax, and as such an example of clever code.	122
30	smr.c	123
31	factorial.c: The use of recursion, rather than iteration, in the computation of a factorial is particularly praised by programmers.	126
32	The comparison two functions, one using recursion, the other one using iteration, intends to show the computational superiority of recursion. (amit, 2012).	127
33	Choose variable names that masquerade as mathematical operators	128
34	Code That Masquerades As Comments and Vice Versa	129
35	All The Names of God, Nick Montfort, 2010, source	140
36	All The Names of God, Nick Montfort, 2010, Selected output .	141
37	Implementation of the Floyd-Warshall algorithm, showing an elegant implementation of a complex theory.	145
38	Example of pseudo-code attempting to reverse-engineer a software system, ignoring any of the actual implementation details, taken from (Nielsen, 2017)	174
39	Example of a program text represented in pseudo code. See 40, 41 and 42 for lower level representations.	191

40	Example of a program text represented in a high level language. See 39 for a higher level representation and 41 and 42 for lower level representations.	192
41	Example of a program text represented in an Assembly language. See 39 and 40 for a higher level representation and 42 for a lower level representation.	192
42	Example of a program text represented in bytecode. See 39, 40 and 41 for higher level representations.	193
43	Example of a program text with syntax highlighting and machine-enforced indentation. See 44 for a functional equivalent, unformatted.	221
44	Example of a program text without syntax highlighting nor machine-enforced indentation. See 43 for a functional equivalent, formatted.	222
45	An example of how source code can be a representation an individual, and can exemplify encapsulation, written in Java.	235
46	An example from the Linux kernel showing that the name and the value of a variable might refer to different things (Linux, 2023).	238
47	Cynical American Preamble, by Michael Carlisle, published in code::art #0 (Brand, 2019)	248
48	Unhandled Love, by Daniel Bezera, published in {code poems} (Bertram, 2012)	254
49	Binary search, implemented by Tim Bray in <i>Beautiful Code</i> highlights variable names as an indicator of the spatial component of the function's performance (Bray, 2007).	256
50	This snippet shows how the spatial extension of the text corresponds to the structural semantics of the code, in the Python programming language.	259

51	This listing includes an execution flow diagram inside the program text itself, testifying to the inherently fragmented and non-linear execution of source code. (Mustacchi, 2019) . . .	261
52	Nested, by Dan Brown and published in {code poems} (Bertram, 2012)	263
53	This listing represents the various steps taken in order to shutdown a HTTP server, and shows multiple aspects of spatio-temporal complexities (WeidiDeng, 2023)	264
54	This header file defines the structure of a program, both in its human use, in its interaction with hardware components, and its decoupling of hardware and software elements.	282
55	genalloc.c, Basic general purpose allocator for managing special purpose memory from the Linux Kernel, displaying examples of source-code spatiality (Muratori, 2014)	287
56	genalloc.c, Basic general purpose allocator for managing special purpose memory from the Linux Kernel, displaying examples of source-code spatiality (Muratori, 2014)	288
57	A textbook example of a fundamental construct in computer science, the linked list. This header file shows all the parts which compose the concept (Kirchner, 2022a).	305
58	A comparison of how to remove an element from a list, with elegance depending on the skill level of the author (Kirchner, 2022b).	307
59	A regular expression matcher by Rob Pike, praised for its elegance and conciseness, but not for its practical utility (Oram & Wilson, 2007)	312
60	A terse example of writing a string to an output in Ruby.	333
61	A verbose approach to writing a string to an output in Java.	333
62	Go proposes an elegant way of ignoring certain variables, with the use of the underscore token.	336

63	JavaScript does not have any built-in syntax to ignore certain variables, resulting in more cumbersome code.	336
64	Iterating in C involves keeping track of an iterating counter and knowing the maximum value of a list beforehand.	342
65	Iterating in Python is done through a specific syntax which abstracts away the details of the process.	342
66	Pointers involve a non-straightforward way to reason about values.	343
67	Ruby syntax does not allow the programmer to directly manipulate pointers	344
68	Nice way to do threads in Go.	346
69	Nice way to do threads in Go.	347
70	Complex way to do threads in C.	347
71	These two range operators are semantically equivalent in Python, but the first is more idiomatic than the second.	351
72	The decorator is the idiomatic way to calculate the sum of the Fibonacci sequence. (Schmitz, 2015)	351
73	Ruby features a lot of syntactic sugar.	352
74	An example of useful comments complementing the source code in Mozilla's layout engine, literally drawing out the graphical task executed by the code.	373
75	A textbook semaphore description in pseudo-code (Arpac-Dusseau & Arpac-Dusseau, 2018)	378
76	Abstracting hardware specific resources via configuration options in an open-source project. (Dicola, 2015)	380
77	A code poem written by Maca in Ruby. (Ortega, 2011)	383
78	The executed output from 77	385
79	Different ways to write a JavaScript function in different functional contexts, with either a focus on pedagogy or skill. 397	

