The role of aesthetics in the understandings of source code

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

ED120 - THALIM

last updated: 2023-08-02

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 scientific 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.

Contents

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 what it aims at accomplishing, and on the contexts in which it operates. This study on source code involves the different groups of people which read and write it, the purposes for which they write it, the programming languages they use to write it, and the natural language they use to speak about it. Most importantly, this thesis focuses on source code as a material and linguistic manifestation of a larger digital ecosystem of software and hardware to which it belongs. Since source code is only one component of 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 demontrate, 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 attempt at providing some responses to our research questions.

1.1 Context

1.1.1 The research territory: code

Most of our modern infrastructure depends, to a more or less dramatic extent, on software systems (Kitchin & Dodge, 2011), from commercial spaces to classrooms, transport systems to cultural institutions, scientific production and entertainment products. Software regulates and automates the 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

```
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

system which powers most of the internet and specialized computation) totalled over 175,000 SLOC, version 4.1 jumped to over 19.5 million lines of code in the span of twenty years (Wikipedia, 2021).

Given such a large quantity of textual mass, one might wonder: who reads this code? To answer this question, we must start 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 computer has to go through in order to perform the action that was described in the source code. *Binary code* is what most of the individuals who interact with computers deal with, in the form of packaged applications, such as a media player or a web browser. They (almost) never have to inquire about 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 fo-

cuses 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 2 (*The freedom to improve the program, and release your improvements to the public, so that the whole community benefits.*) as stated in the FSF's definition of Free Software stipulates that access to source code is required to support these freedoms, a version of source code that is *not concealed*, i.e. readable by both human and machine (Stallman & Free Software Foundation , Cambridge(2002).

In addition to this ability to communicate the ideas latent in it, source code, as an always potentially collaborative object, can be the locus of multiple subjectivities coming together. As Krysa and Sedek state in their definition, *source code is where change and influence can happen*, and where *intentionality and style are expressed* (Fuller, 2008). In their understanding, source code shares some features with natural languages as an intersubjective process (Voloshinov & Bachtin, 1986), and as such is different from the 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 intelligi-

bility of source code, they continue, facilitates its circulation and duplication among programmers. It is this aspect of a socio-technical object that we consider as important as its procedural effectiveness.

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

> *Source code is defined as one or more text files which are written by a human or by a machine in such a way that they elicit a meaningful and succesfully actionable response from 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 interesting 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, addresses 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 thanthrough 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 philosphical 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, ressembling, exemplifying— and their purpose is to communicate a truth about these worlds (Goodman, 1978). In Goodman's view, the arts and the sciences are, in the end, two sides of the same coin. They aim at providing conceptual clarity through formal, systematic means, and the arts—understood here in the broad, Renaissance sense of liberal arts—can and should be, 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 es-

tablishment of two dichotomies: fiction/diction, and constitutivity/conditionality. In *Fiction and Diction* (Genette, 1993), he extends previous conceptions of literature and poetics, from Aristotle to Jakobson, in order to broaden the scope of what can be considered literature, by questioning the conditions under which a text is given a literary status. As such, he establishes the existence of conditional literature alongside constitutive literature: the former gains its status of a literary text from the individual, subjective aesthetic judgment bestowed upon it, while the latter relies on pre-exisiting structures, themes and genres. 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 ex-

amining various program texts directly, and our inquiry into the possibility of multiple aesthetic fields co-existing within source code as a symbol system, asks us to forgo this constitutive definition of an aesthetic work as normative categories within software development. Our focus on sense perception 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.

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

We also distinguish the aesthetic from the beautiful. The latter implies an emotional response and us required 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 limitations is that of text, in its material sense, as mentioned above in our definition of source code. Second, the purpose limitation is that of computable code, rather than computed code: we are examining latent programs, with their reality as texts and their virtuality as actions, rather than the other way around. Executed software and its set of 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 explicited our object of study—the formal manifestations of software under its textual form—we can turn to a review of the research that has alredy been done on the subject, before highlighting some of the 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

```
void leftpad(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.

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, Kerninghan and Plauer publish in 1978 their *Elements of Programming Style* (Kernighan & Plauger, 1978). In it, they focus on how code snippets with a given intent could be rewritten in order to keep the same intent but gain in quality—that is, in readability and understandability. For instance, the program in 2 can be rewritten into the program in 3, which keeps the exact same functionality, but exhibits different formal manifestations.

```c
void leftpad(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.

Why it becomes much clearer, though is not explicited 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 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 As-

sociation 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 (Fakhoury et al., 2019; Guerrouj, 2013), an empirical study of programming style in the efficiency of software teams (Reed, 2010; Coleman, 2018) or on the visual presentation of code in the comprehension process (Marcus & Baecker, 1982) 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 quantitiative 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 cognitive 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 consisten 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 they 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 humanities 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 philos-

ophy. 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 higlight one of the important points of this thesis: the collection of essays in this collected volume focus more often on the graphical output of the software's work from the end-user's perspective than on the textual manifestations of their source (e.g. Nake and Grabowski's essay on the interface as aesthetic event) (Fishwick, 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, discreete/continuous, compilation/interpretation, and flat/stacked languages, clearly acting as a different mode of expression. Second, she draws on a comparison between two main program-

ming 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 as series speech-acts, and leaving aside their structural and post-structural characteristics. Codework is essentially defined by its content and *milieu*, one which focuses on human exchanges and bypasses any involvement of

machine-processing.

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

They consider source code as a located, instantiated presence, understood as a 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 discreete 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 au-

thor 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 78), 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 informatique* (Paloque-Bergès, 2009). This work deploys both linguistic and cultural studies theorists such as Barthes and De Certeau in order to explain these playful acts of source code poetry, along with works of esoteric languages and net.art. 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 intro-

duction 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 Paloque-Bergès's examples, starting from lines of source code in order in order to deduce cultural and social environments and intents through interpreta-

tion. 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 (Pineiro, 2003). In it, he examines the concrete, social and practical justifications for the existence of aesthetics within the software development community. Departing from specific, hand-picked examples such as those featured in Marino's study, his is more of an anthropological approach, revealing what role aesthetics play in a specific community of 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 *intrsumental goodness*, with the aesthetics of code being an attempt to reach excellence in instrumental action. While he carefully lays out his argument by focusing on what programmers actually say, as they exchange about there practice online. 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 the code poets mentioned in some of the works above. 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, quantitiative analysis and, to a lesser extent, qualitative interviews. The research focus is on the effectiveness of aesthetics in code, rather than on unearthing a systematic approach to making code beautiful, even though issues of cognitive friction and understanding, as well as ideals of cleanliness, readability, simplicity and elegance do arise. As such, they form

a 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, as made obvious by code poetry, but such a potential is not assessed through the same empirical lense as the former part of our literature review and only secondarily investigating which of intrinsic features of code can support aesthetic judgments 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 device, 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 explicitely addressed. Depending on which study one reads, one can see code as literature, as architecture, as mathematics or as craft, and there does not seem to be a consensus as to 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 engineer-

ing 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 related to the nature and specificity of software, such a framework should be reflected in an examination of one of the main components of software—source code. The way that this gap has been productively addressed in recent years has primarily been done through an understanding of code as a part of broader socio-technical 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 naming, among other syntactic devices, enable program texts to be expressive. While close-reading will be a useful heuristic for investigating these problems, it will also be necessary to question the unicity of source code, and take into account how it varies across writers and readers and the social groups they constitute. This problem therefore has to be modulated with respect to the 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 aes-

thetics 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 system[2]. 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

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

that we should restrict to any specific genre of program text. As we carry on this study, distinctions will nonetheless arise in our corpus that align with some of the varieties amongst source—for instance, the aesthetic properties of a program text composed of one line of code might be different from those exhbited by a program text made up of thousands of lines code.

We also intend to use source code in both a deductive and an inductive manner. Through our close-reading of program texts, we will highlight some aesthetic features related to its textuality, taking existing source code as concrete proof of their existence. Conversely, we will also write our own source code snippets in order to illustrate the aesthetic features discussed in natural language. 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 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 qual-

ities are only assessed insofar as they relate to the aeshetic manifestations of source code written in those languages.

Meta-texts on source code make up our secondary corpus. Meta-texts are written by programmers, provide additional information, context, 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 langauge, this ability to write comments has been a core feature of any programming language very early on in the history of computing, linking any program text with a potential commentary, whether directly among the source code lines (inline commentary) or in a separate block (external commentary)[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 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

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

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 identifiy 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 explicited 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 be-

tween 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 how 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 effective manifestations to develop our framework in 5. First, we will see how programming languages act as linguistic intefraces 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 lan-

guages 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 in epistemological communication, operating through metaphorical references and structural arrangements at various scales.

We will then turn back to our research questions to suggest some possible answers. The overlap of the aesthetic fields hints at a specifically spatial nature of program texts. Indeed, the specific aesthetics of source code are those of a constant doubling between the specificities of the human (such as natural handling of ambiguity, 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 ambiguity. It is this dynamism, both in terms of *where* and *when* code could be executed, which suggest the use of aesthetics in order to grasp more intuitively the topology and chronology, the state and behaviour of an executed program text.

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 aestethics 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.