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 - 04.10.2021

1 Introduction

This thesis is an inquiry into the formal manifestations of source code and how particular configurations of lines of code allow for aesthetic judgments. The implications of this inquiry will lead us to consider the different ways in which people who read and write code, and through these, we will explore the different ways in which source code can be represented, depending on what it aims at communicating. This study on source code involves the different groups of people which read and write it, the purposes for which they write it, the languages they use to write it, and the language they use to speak about it. Most importantly, this thesis focuses on source code as a material, linguistic manifestation of a larger digital ecosystem of software and hardware to which it belongs. Since source code isn't code, as we will see below, this thesis also aims at studying the reality of written code, rather than its conceptual interpretations.

Starting from pieces of source code, henceforth called *program texts*[1], this thesis will aim at assessing what programmers have to say about it, and attempt to identify one or more specific *aesthetic registers*. This aim depends on two facts: first, source code is a medium for expression, both to express the programmer's intent to the computer[2] and the programmer's intent to another programmer[3]. Second, source code is a relatively new medium of expression, compared to either fine arts or engineering practices. As a recent medium for expression, the development and solidification of aesthetic practices—that is, of ways of doing which do not find their immediate justification in a practical accomplishment—is an ongoing research project in computer science, software development and more recent fields within the digital humanities. Formal judgments of source code are therefore existing and well-documented, and they are related to a need for expressiveness, as we will see in chapter X, but their formalization is still an ongoing process.

Source code thus has ways of being presented which are subject to

aesthetic judgments by programmers; that is, code *has* aesthetics, but it is unclear exactly *which* aesthetics. Indeed, these aesthetic judgments as they exist today rely on different domains to assess source code, as a means to grasp the cognitive object that is software. These draw from metaphors which range from literature, architecture, mathematics and engineering. And yet, source code, while qualified on all of these, source code isn't specifically any of these. Liked the story of the seven blind men and the elephant[4], each of these domains touch on some specific aspect of the nature of code, but none of them are enough to entirely provide a solid basis for the aesthetic judgments of source code. It is at the crossroads of these domains that this thesis locates its work.

The examination of source code, and of the discourses around source code will integrate both the myriad of ways in which source code can exist, and the invariant aspects which underline all diverse approaches of source code. Particularly, we will see how each groups of practicionners tend to deploy references to one particular set of metaphorical references 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*. Relying heavily on Nelson Goodman's work on the languages of art[5], we end on connecting this 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—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 - 15p

1.1.1 The research territory: code

Most of our modern infrastructure depends, to a more or less dramatic extent, on computer systems[6], from commercial spaces to classrooms, transport systems to cultural institutions, scientific production and entertainment products. The complex processes are described in what is called source code, and the number of lines of code involved in running these processes is hard to estimate; one can only rely on disclosures from companies, and publicly available repositories. For instance, all of Google's services amounted to over two billions source lines of code (SLOC)[7], while the 2005 release of the OSX operating system comprised 86 millions lines of code, and while the version 1.0 of the Linux kernel (an operating system which powers most of the internet and specialized computation) totalled over 175,000 SLOC, version 4.1 jumped to over 19.5 million lines of code in the span of twenty years[8].

Who reads this code? To answer this question, we must start diving a little bit deeper into what source code really is.

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

```
a = 4
b = 6

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

compute(a, b)
```

consists in telling the computer to store two numbers in what are called *variables*, then proceeds with describing the *procedure* for adding the dou-

ble of the first terms to the second term, and concludes in actually executing the above procedure. Given this particular piece of source code, the computer will output the number `14` as the result of the operation `(4 * 2)+ 6`. In this sense, then, source code is the requirement for software to exist. If computers are procedural machines, acting upon themselves and upon the world, they need a specification of what to do, and this specification exists in the form of source code.

Source code is here both a requirement and a by-product, since it isn't required anymore once the computer has processed and stored it into a *binary* representation, a series of 0s and 1s which represent the successive states that the computer has to go through in order to perform the action that was described in the source code. *Binary code* is what most of the individuals who interact with computers deal with, and (almost) never have to inquire about, or read its source code. On one hand, then, source code only matters until it gets processed by a computer, through which it realizes its intended function.

On the other hand, source code isn't just about telling computers what to do, but also about a particular economy: that of software development. Software developers are the ones who write the source code and this process is first and foremost a collaborative endeavour. Software developers write code in successive steps, because they add features over time, or they fix errors that have shown up in their software, or they decide to rewrite parts of the source code based on new ideas, skills or preferences. In this case, source code is not used to communicate to the computer what it does, but it is used to communicated 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.

Official definitions of source code straddle the 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) is that of *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*[9]. This definition focuses on the ability of code to be processed by a machine, and mentions little about its readability (i.e. processability by other humans).

On the other hand, the definition of source code provided by the Linux Information Project[1] focuses on source code as *the version of software as it is originally written (i.e. typed into a computer), by a human in plain text (i.e. human-readable, alphanumeric characters).*[10]. 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.*[11]) 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.

In addition to this ability to communicate the ideas latent in it, source code, as an always potentially collaborative object, can be the locus multiple subjectivities coming together. As Krysa and Sedek state it in their

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

definition, *source code is where change and influence can happen*, and where *intentionality and style are expressed*[12]. In their understanding, source code shares some features with natural languages as an intersubjective process[13], and as such is different from the machine language representation of a program, an object which they do not consider source code due to its unilaterality. The intelligibility of source code, they continue, facilitates its circulation and duplication among programmers. It is this aspect of a socio-technical object that we intend to highglight.

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 response from a digital compiler or interpreter, and describe a software system. These text files are the starting point to produce an execution of the system described, whether the very first starting point, or an intermediate representation used for subsequent compilations. These files are called *program texts*.

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

forms of source code, such as visual programming languages or biological computation, stand outside the scope of this study and should be investigated in subsequent work.

As for the term *program text*, it is chosen in order to highlight the dual nature of source code: that of a computational artefact to be formally processed and unambiguously understood, and that of a open-ended, multi-layered document, in the vein of Barthes' distinction between a text and a work[14].

1.1.2 Beautiful code

Under this definition of source code textually represented, we now turn to the existence of the aesthetics of such *program texts*. To contextualize this existence, we first need to touch upon the history and practice of software development. As an economic activity, software development came from a bottom-up dynamic, a *de facto* activity which was not expected in the early days of computing, where most of the work was divided between mathematics and engineering. Its earliest manifestation can be found in the physical rewiring process of mainframes in order to perform a specific computation, something more akin to firmware than to software. These rewiring tasks were done by mostly female assistants, under the direction of mostly male mathematicians[15], and considered a simple translation task which did not need any particular attention, or any particular skill. The recognition of software engineering as its own field came as its unique domain of expertise was required in larger engineering projects—for instance, the term *software engineering* was coined in the late 1960s by Margaret Hamilton and her team as they were working on the Apollo 11 Lunar Module software[16]. In the same decade, the first volume of *The Art of Computer Programming*, by Donald Knuth, addresses directly both the existence of programming as an activity separate from both mathematics and engineering, as well as an activity with an "artistic" dimension[17]. The first volume

opens on the following paragraph:

> The process of preparing programs for a digital computer is especially attractive, not only because it can be economically and scientifically rewarding, but also because it can be an aesthetic experience much like composing poetry or music. This book is the first volume of a multi-volume set of books that has been designed to train the reader in the various skills that go into a programmer's craft.[17]

Considered one of the most canonical textbooks in the field, *The Art of Computer Programming* lays out two important aspects of programming: that it can be an aesthetic experience and that it is the result of a craft, rather than of a highly-formalized systematic process. Some of the aeshetic references related to source code are related to its writing and reading being a craft-like activity[18].

Craftsmanship as such is an essentially 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*[19], in which ways of doing are prescribed in a top-down fashion. It is hard to formalize, and the development of expertise in the field happens through practice as much as through formal education[20]. The domain of craft is also one in which function and beauty exist in an intricate, embodied relationship, based on subjective qualitative standards rather than strictly external measurements, with the former rarely being explicitly stated[21].

Approaching programming (the activity of writing and reading code) as a craft[22] connects to the multiple testimonies of encountering beautiful code, some of which have made their ways into edited volumes or monographs[23, 24, 25]. Additionally, informal exchanges among programmers on forums, mailing lists, blog posts and code repositories[2] of-

[2] Insert references about the annex here

ten mention beautiful code, either as a central discussion point or simply in passing. These testimonies, from textbooks to online posts, 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. However, the formalization of an aesthetics of source code first requires a formalization of the concept of *aesthetics*.

There is a long history of aesthetic philosphical inquiries in the Western tradition, from beauty as the imitation of nature[3], moral purification[4], cognitive perfection[5], sensible representations with emotional repercussions[6]. The common point of these definitions is that of *sensual manifestation*, that is the set of visible forms which can enable an *aesthetic experience*, a cognitive state of pleasure relying on, amongst others, an object, a sense of unity and of discovery[26].

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[5] and Gérard Genette's distinction between fiction and diction[27]. 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 art to communicate. This communication process happens through

[3] Plato

[4] Aristotle, Poetics; Kant, Critique of the Power of Judgment

[5] Leibniz, Ars Combinatoria

[6] Baumgarten, Aesthetics

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[5]. 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[28]. In his 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 can and should be approached with the same rigor as the sciences. Programming, with its self-proclaimed craft status, stands equally across the line divind arts and sciences.

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

With this analytical framework allowing us to analyze the matter at hand—program texts as composed by a symbol system with a epistemic purpose—we turn to a more literary perspective on aesthetics. Genette's approach to literature, *the art of language*, results in the establishment of two dichotomies: fiction/diction, and constitutivity/conditionality. In his eponymous work[27], 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 broadening the conditions under which a text is given a literary status. As such, he establishes the existence of conditional literature alongside constitutive literature: the former gains its status of a literary text from the individual, subjective aesthetic judgment bestowed upon it, while the latter relies on pre-exisiting structures, themes and genres. This approach paves the way for an extending of the domain of literature[29], and a more subtle understanding of the aesthetic manifestation in textual works.

Genette 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 as a purely functional text—i.e. what the source code ultimately does. 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. Since we focus on the written form of source code, and not on the type of its purpose, an attention to diction will be the entry point of this thesis.

A first approach to source code could be *constitutive*, in Genette's terms: a given program text could be considered aesthetically pleasing because the software it generates abides by some definitions of being aesthetically pleasing[7], or because the software itself is considered a piece of art, shown in exhibitions and sold in galleries. However, our empirical approach to source code aesthetics, by examining various program texts directly, and our inquiry into the possibility of multiple aesthetic registers co-existing within source code as a symbol system, asks us to forgo this constitutive definition of an aesthetic work as normative categories which do not yet

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

exist within software development. Our focus on sense perception within aesthetics starts then 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*[8].

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

So, following Genette's re-asking of the Goodman's question of *When is art?* rather than the historical *What is art?*, we can now proceed with our understanding of aesthetics as a set of physical manifestations which can be grasped by the senses, whether "the movement of a light, the brush a fabric, the splash of a color"[30], which aim at enabling a cognitive, communicative purpose, and which are not exclusively constituted by pre-existing categories.

These relations between source code and aesthetics have been addressed by academic studies through different, separate dynamics.

1.1.3 Literature review

A literature review on this topic must address the dualistic nature of studies on source code. Reminiscent of C.P. Snow's distinction of two cultures, research can be clearly divided 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

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

aesthetics have been explored until now, and will invite us to address the remaining gaps.

Most technical literature, starting from *The Art of Computer Programming*, acknowledges the role that aesthetics have to play in the writing and reading of program texts. Along with the positions of Knuth and Djikstra regarding the importance of paying attention to all aspects of programming practice, beyond strictly mathematical and engineering requirements, Kerninghan and Plauer publish in 1978 their *Elements of Programming Style*[31], focusing on how code snippets with a given intent could be rewritten in order to keep the same intent but gain in quality—that is, in readability and understandability. For instance, the following program:

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

can be rewritten as:

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

which keeps the exact same functionality, but becomes much clearer. Why it becomes much clearer, though, is thought to be a given for the reader, and not explicited by the authors in terms ofc concepts such as cognitive surface, repleteness of a symbol system or representation of the idea at play (casting an integer to a character, rather than individually checking for each integer case). However, the authors do employ terms which will form the basic of an aesthetics of software development, such as clarity, simplicity, or expressiveness; still, there are no overarching prin-

ciples deployed to systematize the approach, only examples of such principles.

While Kernighan and Plauer do not directly address the relationship of source code and aesthetics, this is something that Peter Molzberger undertakes five years later through an empirical, qualitative study aimed at highlighting the place aesthetics play in an expert programmer's practice[32]. 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 phenomena 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 this switch in scales and directions of reading, or a conception of code as mathematics does not explain the need for a personal touch when writing source code[32].

In the context of formal academic research, such as the IEEE or the Association for Computing Machinery (ACM), research then focuses on how to quantitatively assess a given quality of source code either through a social perspective on the process of writing[33], a semantic perspective on the lexicon being used[34, 35], an empirical study of programming style in the efficiency of software teams[36] or on the visual presentation of code in the comprehension process[37]. These focus on the connection of aesthetics with the performance of software development—beautiful code might be related to a good end-product.

In parallel, the development of software engineering as a profession has led to the publication of several books of specialized literature, taking a practical approach to writing good code, rather than a scientific one. Robert C. Martin's *Clean Code*'s audience belongs to the field of business and 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, and was

followed by a number of additional publications on the same topic and with the same approach[38, 39, 40]. Here, these provide an interesting counterpoint to academic research on quality code by relying on different traditions to explain why the way code is written is important.

Technical and engineering literature, then, establish the existence of and need for aesthetics, presented as formal properties which then constitute *quality code*. The methodology in these studies is either empirical, in the case of academic articles, looking at lines of code, or interviewing programmers in order to draw conclusions regarding this relationship between formal properties and quality, while earlier 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 high-level programmers are invited to pick a piece of code and explain why they like it, sometimes commenting it line by line[23]. This very concrete, empirical inquiry into what makes source code beautiful does no, however, include a strong enough conclusion as to what *actually* makes code beautiful. 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, the group of those who write and read source code is far from being homogeneous, and can actually be grouped into distinct categories—e.g. academics, tinkerers or artists—with distinct practices and standards[41]. It is not considered 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, in which inquiries into the nature of computation, from an ontological, epistemological and ethical points of view. These are useful both in the meta positioning they take regarding computer science as they well as how they show that issues of representation, interpretation and implementation

are complicated issues 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—very few references to computer science as having any kind of relation with literature[42]. Another, narrower 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, focus on its creative and relationship to patters, a subject we will investigate more in chapter X. Finally, Brian Cantwell-Smith's introduction to his upcoming *The Age of Significance: An Essay on the Origins of Computation and Intentionality* touches upon these similar ideas of intentionality by suggesting both that computation might be more productively studied from a humanities or artistic point of view than form a strictly scientifc point of view[43]. These philosophical inquiries into computation mention aesthetics mostly on the periphery, but nonetheless challenge the notion of computation as strictly functional, and suggest additional perspectives on the topic, 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. In particular, each of these monographs, edited volumes, catalog articles, book chapters or PhD theses, engage with code in its multiple intricacies. Source code excerpts, programming environments and languages form a part of each of these works as primary sources and are considered as more often than not as text to be read and examined closely, and in that sense all offer the same foray into code-as-text as my work intends to.

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 than on the textual manifestations of their source (e.g.

Nake and Grabowski's essay on the interface as aesthetic event)[44]. As for most studies in Computer Science conferences, the focus is on Human-Computer Interaction (HCI) as the art and science of presenting visually the output and affordances of a running program. While a vast and complex field, this is not the topic of this thesis: rather than focusing on the aesthetics of the computed and executed, it focuses on the aesthetics of the computable (texts).

The following works, because of their dealing with source code as text, and due to the background of their authors in literature and comparative media studies, incorporate some aspect of literary theory and criticism, and authors such as N. Katherine Hayles, Maurice J. Black and Alan Sondheim rely on it as their principal perspective. Black, in his PhD dissertation *The Art of Code*[45] 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, rather than formal, definition of a source code aesthetic.

At this point, it seems that Black operates this study in only one of two directions: by establishing 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. In summary, Black provides

a first study in code as a textual object and as a textual practice, but does not address what makes code poetry different in its writing, reading and meaning-making than natural-language poetry.

N. Katherine Hayles, in her book *My Mother Was A Computer: Digital Subjects and Literary Texts*[46], and particularly in the *Speech, Writing, Code: Three Worldviews* 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[47], and attempts to qualify this worldview both in opposition to Saussure's *parole* and Derrida's *trace*, following cybernetics and media studies thinkers such as Friedrich Kittler and Mark B. Hansen. Specifically, she introduces the idea of a Regime of Computation, which relies on the conceptual specificities of code-based expression (among which: depth, dynamism, fragmentation, etc.). Source-code specific contributions touch upon literary paradigms and cognitive effect. First, she highlights the way code recombines some traditional dialectics of literary theory, namely paradigmatic/syntagmatic, discreete/-continuous, compilation/interpretation, and flat/stacked languages. Second, drawing 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 the structure and the meaning of programming texts, depending 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. She then locates this materiality in the embodiment of users and readers, along with authors such as Mark Hansen[48], Bernadette Wegenstein[49] and Pierre Lévy[22]. Beyond the brief acknowledgment that she has of the political and economical conditions of software development and their impact on electronic texts, she also stops short of considering programming languages in their varieties, and the material appara-

tuses which support them (documentation, architectures, compilers, tutorials, conferences and communities). In the vein of this material approach, a conception of programming languages as material seems like a possible avenue for looking into the formal possibilities they afford.

Alan Sondheim's essay *Codework*[50], as the introduction of the American Book Review issue dedicated to this specific form, provides another aspect of poetry that which integrates source code as a creole language emerging from the interplay of natural and machine languages. However, this specific aspect of literary work integrates the surface of code rather than with its structure and therefore provides more insight in the anthropology of how humans represent code through speech, rather than representing speech through code (i.e. focusing first and foremost on the structural uniqueness of programming languages). This presents a somewhat postmodern view of programming languages, forcing them upon a relational, mutable conception of language as as series speech-acts, and leaving aside their structural and post-structural characteristics. Codework is essentially defined by its content and *milieu*, one which focuses on human exchanges and bypasses any involvement of machine-processing.

Another perspective on the relationship between speech and code is explored by Geoff Cox and Alex Mclean in *Speaking Code: Coding as Aesthetic and Political Expression*[51]. They establish an interpretation of reading, writing and executing source code as a speech-act, extending J.L. Austin's theory to a broader political application by combining Arendt's approach of human activities and labor[52], from which coding is seen as the practice of producing laboring speech-acts.

They consider source code as a located, instantiated presence, understood as a politically semantic object affecting the multiple economic, 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. However, on a more formal

level, the authors often illustrate the points they are working through, or begin developing those points, with snippets of code written by either McLean or established software artists, thus engaging with details of source code and taking a step away from the dangers of fetishizing code, or *sourcery*[4]. 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), highlighting the possible intertextuality of program texts and natural texts. Still, we can find a limitation of the almost exclusive focus on speech and live aspect of code, side-stepping the particular grammatical features of that speech.

Away from the cultural relevance of code as developed by Cox and McLean, Cramer focuses on the cultural history of writing in computation, tying our contemporary fascination with source code into an older web of historical attempts at integrating combinatorial practices from Hebraic texts to Leibniz's universal languages[53]. 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), itself reminiscent of Simondon's definition of a technical object's essence[54]. By relocating it between magic and reality, code is no longer just arbitrary symbols, or machine instructions but also ideal execution, a set of discreete forms which relate to the totality of the world. Once formal execution is considered a cosmogonical force, it becomes synonymous with performative execution, through which it ties back to cultural practices throughout the ages, within both religious and scientific contexts.

Cramer extracts five axes along which to apprehend code-based works: totality/fragmentation, rationalization/occultation, hardware/software, syntax/semantics, artificial/natural language. While all these axes overlap each other, it is the *syntax/semantics* axis which aligns most with this research, hypothesizing that it is possible to touch upon these other thematical axes through it. 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. Indeed, 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.*[53]

It therefore seems that there is a relationship between the formal disposition of source code within program texts and the cultural communities composed of the writers and readers of these program texts. As we've seen, code has a social component, insofar as it operates as an expressive medium between various subjects.

Adrian MacKenzie approaches source code, as part of a broader inquiry on the nature of software, through this social lens in *Cutting Code: Software and Sociality*[55]. The author focuses on a relational ontology of software: it is defined in how it acts upon, and how it is being acted upon by, external structures, from intellectual property frameworks to design philosophies in software architectures; it only provides an operational definition—software is what it does. His analysis of source code poetry focuses on famous Perl poems, Jodi's artworks and Alex McLean's `forkbomb.pl`, still focusing on the executability of code as its dominant feature, dismissing Perl poetry as *"a relatively innocuous and inconsequential activity"*[55]—an activity which programmers nonetheless spent most of their days doing. While software could indeed be a *patterning of social relations*, these social relations also take place through linguistic combinations in program texts.

Focusing deeper on the category of code poetry, Camille Paloque-Berges published, a couple of years later, *Poétique des Codes sur le réseau informatique*[56]. 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 esoteric languages and net.art.

While the first chapter focuses on digital literature as the result of executed code in order to develop a heuristic to approach source code, and while the third and last chapter focuses on the means of distribution of these works, particularly on the development of net.art, 1337 5p43k and codeworks, the second chapter is the most relevant to our research focus. In it, Paloque-Berges provides an introduction of creative acts in source code on both a conceptual level (drawing from Hayles and Montfort) and on a technical, syntactical level. She looks at specific programming patterns and practices (”hello world”, quines), technical syntax (e.g. `$`, `@` as Perl tokens for expressing singular or plurals) and cultural paradigms (De Certeau's tactics vs. strategies), as she attempts to highlight the specificities of source code for aesthetic manifestationsand she invites further work to be done in this dual vein of close-reading and theoretical contextualization, beyond specific, heightened manifestations such as Perl poetry.

Honing on a minimal excerpt, *10 PRNT CHR$(205.5+RND(1)) : GOTO 10;*[57], is a collaborative work examining the cultural intertwinings of a single line of code, through hardware, language, syntax, outputs and themes. The whole endeavour is one rigorous close-reading of source code, in a clearly deductive fashion, working from the words on the screen and elaborating the context within which these words exist, in order to establish the cultural relevance of source code, as related to the syntax, hardware and cultural context in which these words exist. While the study itself, being a close-reading of only one work, and particularly a *one-liner*, itself a specific genre, is restricted in terms of broad aesthetic statements, it does show how it is possible to talk about code not as an abstract construct but as an , concrete reality. Particularly interesting is the section dedicated to the history of the BASIC programming language, and how particular languages afford particular statements and actions better in a given historical context, a point often glossed over in other studies.

A current synthesis of these approaches, Mark C. Marino's *Critical Code Studies*[58] and the eponymous research field it initiates focuses on close-

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

Taking a step back, Warren Sack's *The Software Arts*[59] historicizes the history of software development as an epistemological practice, rather than a strictly economic trade. Connecting some of the main components of software (language, algorithm, grammar), he demonstrates how these are rooted in a liberal arts conception of knowledge and practice, particularly visible as a parallel to Diderot and D'Alembert's encyclopedic attempt at formalizing craft practices. By examining this other, humanistic, tradition in parallel with the traditionally acknowledged scientific one, Sack shows the multiple facets that code and software can support. Starting from the concept of "translation" as an updated version of Manovich's "transcoding", Sack analyzes what is being translated by computing, such as analyses, rhetoric and logic, byt doesn't however address the nature of the process in which these concepts are translated—algorithms as ideas, but not as texts.

This activity of programming as craft, already acknowledged by

programmers themselves, is further explored in Erik Pineiro's doctoral thesis[60]. In it, he examines the concrete, social and practical justifications for the existence of aesthetics within the software development community. Departing from specific, hand-picked examples such as those featured in Marino's study, his is more of an anthropological approach, revealing what role aesthetics play in a specific community of practicioners. Outlining references to ideas such as *cleanliness*, *simplicity*, *tightness*, *robustness*, amongst others, as the aesthetic ideals that programmers aspire to, he does not however summon any specific aesthetic register (whether from literature, mathematics, craft or engineering). Rather frames it in terms of *intrsumental goodness*, with the aesthetics of code being an attempt to reach excellence in instrumental action. While he carefully lays out his argument by focusing on what (a certain group of) programmers actually say, instead of what they might be saying, there remains two limitations: it is not clear how source code as textual material can afford to reach such aesthetic ideals, and whether or not these aesthetic ideals apply to other groups of writers of code, such as the code poets mentioned in some of the works above.

This literature review allows us to have a better grasp of how the relationship between source code and aesthetics—in its sense of formal manifestation and organization—has been studied, both from a scientific and engineering perspective, and from a humanities perspective.

In the former perspective, 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 aesthetic 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 nonetheless form a good starting ground. 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 unclear, straddling different lines while not aligning clearly with either science, engineering or arts—it seems that software is indeed something different.

As for the latter perspective on source code and aesthetics, that of the humanities, the highlight is predominantly on literary heuristics or on social dynamics, and the details and examples of the actual code syntax and semantics are often omitted. Aesthetic aspects of a literary or cultural nature are being explored in code: there is a potential for beauty and art in source code, but such a potential is not assessed through the same lense as the former part of our literature review and only secondarily investigating which of intrinsic features of code can support aesthetic judgments.

Still, some recent studies, such as Paloque-Bergès's, Montfort et. al's, Cox and McLean's and Marino's, do engage directly with source code examples, and these constitute important landmarks for a code-specific aesthetic theory and methodology, either as poetic language, speech-act, or critical commentary. In broad terms, source code is taken as a unique literary device, but it remains unclear exactly in which aspects it is different from both natural languages and low-level machine languages, and how this literary aspect relates to the effective, mathematical and craft-like nature of source code considered in the computer science and engineering literature.

- ~~start with a very basic description of what code/source code/software is~~
- ~~quick history of software development~~
- ~~demonstrate that program texts are a thing then that programmers~~

~~know there is beautiful code.~~

- ~~connect the "textual manifestation" aspect with my definition of aesthetics.~~
- then a literature review on how these issues of aesthetics+code are addressed

1.1.4 The definitions

- ~~Aesthetics~~
- ~~Source Code~~
- ~~Program Text~~

1.1.5 Boundaries

that which i shall not touch (visual programming, bio programming, based on the fact that I'm starting from a literary point of view)

but once we have laid out all of this, then it becomes possible to highglight the gaps:

- the assumption of the literarity of code, rather than architecture
- the lack of theoretical framework explaining why some code is beautiful and why other isn't
- the lack of *proof*, which is showing code
- the apparent assumption that aesthetics have nothing to do with source code

1.2 Problem - 10p

start by establishing, based on what was said previously, the *niche* that i will occupy

- the fetishization of code
- the multiplicity of registers
- the reason why it matters (connecting beautiful and useful)

then really establish the significance of this study: beauty and function, and conclude on my research questions

1.3 Methodology - 10p

- reading code
- reading discourses
- reading aesthetic theory based from these discourses
- also the theoretical frameworks (maybe move goodman and genette to here)

1.4 Roadmap - 6p

list all chapters with a brief overview for each.

1.5 Connecting back to the wider world - 2p

and readership

References

[1] Francoise Detienne. *Software Design – Cognitive Aspect*. Springer Science & Business Media, December 2012.

[2] Edsger W. Dijkstra. Chapter I: Notes on structured programming. In *Structured programming*, pages 1–82. Academic Press Ltd., 1972.

[3] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs - 2nd Edition*. Justin Kelly, 1979.

[4] Wendy Hui Kyong Chun. On "Sourcery," or Code as Fetish. *Configurations*, 16(3):299–324, 2008.

[5] Nelson Goodman. *Languages of Art*. Hackett Publishing Company, Inc., Indianapolis, Ind., 2nd edition edition, June 1976.

[6] Rob Kitchin and Martin Dodge. *Code/Space: Software and Everyday Life*. The MIT Press, 2011.

[7] @Scale. Why Google Stores Billions of Lines of Code in a Single Repository, September 2015.

[8] Linux kernel, October 2021. Publication Title: Wikipedia.

[9] Mark Harman. Why Source Code Analysis and Manipulation Will Always be Important. In *2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*, pages 7–19, September 2010.

[10] Source code definition by The Linux Information Project.

[11] Richard Stallman and Mass) Free Software Foundation (Cambridge. *Free software, free society : selected essays of Richard M. Stallman*. Boston, MA : Free Software Foundation, 2002.

[12] Matthew Fuller, editor. *Software Studies: A Lexicon*. The MIT Press, Cambridge, Mass, April 2008.

[13] V. N. Voloshinov and Michail M. Bachtin. *Marxism and the Philosophy of Language*. Harvard University Press, 1986.

[14] Roland Barthes. *Le bruissement de la langue: essais critiques IV*. Seuil, Paris, 1984.

[15] Wendy Hui Kyong Chun. On Software, or the Persistence of Visual Knowledge. *Grey Room*, 18:26–51, January 2005.

[16] David A. Mindell. *Digital Apollo: Human and Machine in Spaceflight*. MIT Press, September 2011.

[17] Donald E. Knuth. *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., USA, 1997.

[18] Edsger W. Dijkstra. "Craftsman or Scientist?". In Edsger W. Dijkstra, editor, *Selected Writings on Computing: A personal Perspective*, Texts and Monographs in Computer Science, pages 104–109. Springer, New York, NY, 1982.

[19] Michel de Certeau, Luce Giard, and Pierre Mayol. *L'invention du quotidien*. Gallimard, 1990.

[20] Richard Sennett. *The Craftsman*. Yale University Press, 2009.

[21] David Pye. *The Nature and Art of Workmanship*. Herbert Press, illustrated edition edition, July 2008.

[22] Pierre Lévy. *De la programmation considérée comme un des beaux-arts*. Textes à l'appui. Anthropologie des sciences et des techniques. Éd. la Découverte, Paris, 1992.

[23] Andy Oram and Greg Wilson, editors. *Beautiful Code: Leading Programmers Explain How They Think*. O'Reilly Media, Beijing ; Sebastapol, Calif, 1st edition edition, July 2007.

[24] Vikram Chandra. *Geek Sublime: The Beauty of Code, the Code of Beauty*. Graywolf Press, September 2014.

[25] Richard P. Gabriel. *Patterns of Software: Tales from the Software Community*. Oxford University Press, 1998.

[26] Monroe C. Beardsley. The Aesthetic Point of View*. *Metaphilosophy*, 1(1):39–58, 1970.

[27] Gérard Genette. *Fiction & Diction*. Cornell University Press, 1993.

[28] Nelson Goodman. *Ways Of Worldmaking*. 1978.

[29] Alexandre Gefen and Claude Pierre Perez. 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*, September 2019.

[30] Jacques Ranciere. *Aisthesis: Scenes from the Aesthetic Regime of Art*. Verso, London ; New York, 1st edition edition, June 2013.

[31] Brian W. Kernighan and P. J. Plauger. *The Elements of Programming Style, 2nd Edition*. McGraw-Hill, New York, 2nd edition edition, January 1978.

[32] Peter Molzberger. Aesthetics and programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '83, pages 247–250, New York, NY, USA, December 1983. Association for Computing Machinery.

[33] Brandon Norick, Justin Krohn, Eben Howard, Ben Welna, and Clemente Izurieta. 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, page 1, New York, NY, USA, September 2010. Association for Computing Machinery.

[34] Sarah Fakhoury, Devjeet Roy, Sk. Adnan Hassan, and Venera Arnaoudova. Improving source code readability: theory and practice. In *Proceedings of the 27th International Conference on Program Comprehension*, ICPC '19, pages 2–12, Montreal, Quebec, Canada, May 2019. IEEE Press.

[35] Latifa Guerrouj. Normalizing source code vocabulary to support program comprehension and software quality. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 1385–1388, San Francisco, CA, USA, May 2013. IEEE Press.

[36] David Reed. Sometimes style really does matter. *Journal of Computing Sciences in Colleges*, 25(5):180–187, May 2010.

[37] Aaron Marcus and Ronald Baecker. On The Graphic Design of Program Text. May 1982.

[38] Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts, and Erich Gamma. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, Reading, MA, 1st edition edition, July 1999.

[39] Inke Arns. Code as performative speech act. *Artnodes*, 0(4), May 2005.

[40] Andrew Hunt and David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Professional, Reading, Mass, 1st edition edition, October 1999.

[41] Brian Hayes. *Cultures of Code*. February 2017.

[42] William J. Rapaport. Philosophy of Computer Science: An Introductory Course. *Teaching Philosophy*, 28(4):319–341, 2005.

[43] Brian Cantwell Smith. *On the Origin of Objects*. A Bradford Book, Cambridge, Mass., reprint edition edition, January 1998.

[44] Paul Fishwick. Aesthetic Programming. February 2001.

[45] Maurice Joseph Black. The art of code. *Dissertations available from ProQuest*, pages 1–228, January 2002.

[46] N. Katherine Hayles. *My Mother Was a Computer: Digital Subjects and Literary Texts*. University of Chicago Press, March 2010.

[47] Walter J. Ong. *Orality and Literacy: 30th Anniversary Edition*. Routledge, London, 3 edition, September 2012.

[48] Mark B. N. Hansen. *Bodies in Code: Interfaces with Digital Media*. Routledge, New York, September 2006.

[49] Bernadette Wegenstein. Bodies. In *Critical Terms for Media Studies*. University of Chicago Press, March 2010. Google-Books-ID: eb4HDw0CklEC.

[50] Alan Sondheim. Introduction: Codework. *American Book Review*, 22(6), October 2001.

[51] Geoff Cox and Christopher Alex McLean. *Speaking Code: Coding as Aesthetic and Political Expression*. MIT Press, 2013.

[52] Hannah Arendt. *The human condition*. University of Chicago Press, Chicago, 2nd ed. / introduction by margaret canovan. edition, 1998. Open Library ID: OL23240850M.

[53] Florian Cramer. *Words Made Flesh*. Piet Zwart Institute, 2003.

[54] Gilbert Simondon. *Du mode d'existence des objets techniques*. PhD thesis, Aubier et Montaigne, Paris, 1958. OCLC: 410239717.

[55] Adrian Mackenzie. *Cutting Code: Software and Sociality*. Peter Lang, 2006.

[56] Camille Paloque-Bergès. *Poétique des codes sur le réseau informatique*. Archives contemporaines, 2009.

[57] Nick Montfort, Patsy Baudoin, John Bell, Ian Bogost, and Jeremy Douglass. *10 PRINT CHR$(205.5+RND(1)); : GOTO 10*. The MIT Press, illustrated edition edition, August 2014.

[58] Mark C. Marino. *Critical Code Studies*. Software Studies. MIT Press, Cambridge, MA, USA, March 2020.

[59] Warren Sack. *The Software Arts*. The MIT Press, April 2019.

[60] Erik Pineiro. *The aesthetics of code : on excellence in instrumental action*. PhD Thesis, KTH, Superseded Departments, Industrial Economics and Management., 2003.