Le rôle de l'esthétique dans la compréhension du code source

Pierre Depaz
sous la direction d'Alexandre Gefen (Paris-3)
et Nick Montfort (MIT)

ED120 - THALIM

Septembre 2023

1 Introduction

En tant que texte dont le but même est de disparaître (transformé en changements de courant électrique), le code source est un objet hybride, autant description qu'action, aux interlocuteurs multiples. Il peut être communication entre humain et humain, entre humain et machine, ou entre machine et machine. Cependant, en tant que création humaine, il est possible de s'interroger sur les modalités de ses manifestations, notamment en termes d'expressivité. Si l'expressivité fonctionnelle d'un code source implémentant un algorithme est indéniable, l'expressivité artistique d'un texte de code source demeure élusive. Quelle est donc la place de l'esthétique, d'une manifestation formelle sensuellement plaisante, dans l'écriture ou la lecture du code source? Le code source, basé sur un système syntaxique similaire au language naturel, se révèle être un système de communication aussi bien d'humain à humain que d'humain à machine. Il

semble néanmoins être principalement destiné les humains, et secondairement aux les machines (Abelson et al., 1979). Une fois que la condition principale d'existence du code source (sa validité d'exécution) est satisfaite, une condition secondaire semble être non pas sa beauté mais sa compréhensibilité.

Se pose donc la question d'un texte dont les manifestations formelles sont vouées à disparaître, et dont la lecture n'est qu'un processus collatéral de son exécution. Donald Knuth, dans son oeuvre *The Art of Computer Programming* (Knuth, 1997), commence par établir que la programmation (l'écriture et la lecture du code source) est bel et bien un art. Cette déclaration, dans les premières pages du premier volume, n'est pourtant pas réitérée ni élaborée dans les autres volumes du monographe. Si écrire du code peut être un art, certains codes sources peuvent donc exhiber des propriétés esthétiques, mais ces dernières sont rarement explicitées par la littérature sur le sujet.

La problématique principale de ce projet de recherche est donc celle du *rôle de l'esthétique dans les compréhensions du code sources*. Il s'agit tout d'abord de mettre à jour les propriétés esthétiques propres au code source, et d'identifier comment celles-ci permettent de faire sens. En plus d'une approche empirique des codes sources en eux-mêmes, il s'agit de mettre à jour de quelles manières ces propriétés sont similaires à d'autres domaines de créations mobilisés par les programmeurs, notamment au domaine de la littérature, de l'architecture, des mathématiques et de l'ingénierie. En examinant ces relations, cette thèse s'inscrit donc dans un travail de recherche sur la dimension cognitive du phénomène esthétique.

Une approche rapide de l'état de l'art sur ce sujet révèle deux tendances séparées: d'une part, la littérature en informatique et ingénierie prend en compte l'évidence de l'existence d'une esthétique du code source, du point de vue la productivité de ceux et celles qui écrivent du code, et d'un point de vue cognitif de la compréhension des bases de code (Oram & Wilson, 2007; Cox & Fisher, 2009; Gabriel & Goldman, 2001; Martin, 2008; Deti-

enne, 2001; Weinberg, 1998). Néanmoins, ces études considèrent une esthétique comme un phénomène donné, aux conséquences plus ou moins mesurables, sans pour autant systématiser leur approche au niveau conceptuel.

D'autre part, les recherches en sciences humaines traitent de l'interaction entre esthétique et code, tout en restant majoritairement attachées à une conception abstraite et désincarnée du "code", élaborant une esthétique du digital sans pour autant rentrer dans les détails des codes sources eux-mêmes (e.g. il ne semble pas pertinent qu'ils soient écrits en Perl, ou Python, ou Ruby, etc.) (Cramer, 2019; Hayles, 2010; Mackenzie, 2006; Lévy, 1992). Il existe cependant un certain nombres d'ouvrages de sciences humaines approchant la question matérielle du code de manière directe, que ce soit au niveau culturel (Montfort et al., 2014), politique (Cox & McLean, 2013) ou sociologique (Paloque-Bergès, 2009) Deux travaux doctoraux se sont précédemment intéressés à l'esthétique du code source (Black, 2002; Pineiro, 2003), mais se limitent à des communautés d'écriture du code bien précises, sans examiner les codes sources eux-mêmes. C'est au sein de cette approche sur les manifestations du code source que cette recherche s'inscrit.

2 Méthodologie

L'approche principale de ce projet est premièrement empirique. Il s'agit d'examiner les codes sources eux-mêmes, ainsi que les discours de ceux et celles qui les écrivent et les lisent. Pour cela, je m'appuie sur les travaux de Kintsch et van Dijk et leurs études des stratégies de compréhension du discours (Kintsch & van Dijk, 1978), l'approche métaphorique considérée comme une stratégie de compréhension. Le code source étant un texte, autour duquel sont produits des discours.

Il a d'abord s'agit de faire émerger une variété d'individus écrivant et

lisant du code, que j'ai regroupé en quatre catégories (Hayes, 2015). Les *développeurs* sont les individus qui écrivent du code dans un contexte économique, à but fonctionel et soutenable à long-terme; ils travaillent sur des bases de code extensives et souvent en collaboration. Les *artistes* sont les individus écrivants du code dans un cadre expressif et artistique, tels que des *code poems*; leur code est destiné principalement à être lu par un public, et seulement de manière secondaire à être exécuté. Les *hackers* sont les individus qui écrivent du code en tant que solution technique unique, idiosyncratique et hautement contextuelle; leur code est destiné à être exécuté par une machine particulière, et de manière secondaire par des humains. Enfin, les *académiques* sont les individus qui écrivent du code afin d'illustrer des concepts de computation, représentant des abstractions plutôt que des usages concrets.

Ces quatres groupes ont des limites poreuses, et un code source peut, souvent, appartenir à plusieurs de ces catégories en même temps (une référence artistique dans un contexte commercial, un hack présenté comme oeuvre d'art, ou une approche particulière au sein d'une base de code commerciale, etc.).

La constitution du corpus de cette recherche s'est faite principalement par la consultation de ressources en ligne. En effet, la plupart des bases de code existe sur des sites d'aggrégations (GitHub, BitBucket, etc.) ou sous formes d'extraits (*snippets*) présentés et commentés dans des forums, sur des sites personnels, ou des sites de question/réponse (e.g. StackOverflow, Quora). Cette constitution du corpus inclue non seulement ces sources primaires (le code en soi), mais également des sources secondaires (le commentaire par un auteur, ou par un public, justifiant de l'aspect esthétique d'un code présenté). De cette manière, je considère la valeur esthétique d'un code source comme étroitement lié au jugement de groupe émis à son encontre.

Ma métholodologie consiste en une approche interprétative du corpus constitué, dans un double-mouvement. D'une part, les références et dis-

cours des individus écrivants du code sont pris en compte afin de constituter un ensemble de standards esthétiques: ces standards sont des manifestations concrètes du code permettant une facilitation de la compréhension du *propos* du code (propos prescriptif, ou effectif). D'autre part, les standards exhibés sont appliqués à d'autres bases de code afin de vérfier leur applicabilité. Ã travers la multiplicité de ces standards, il s'agit donc d'identifier les stratégies métaphoriques de compréhensions des lecteurs de code.

Cette examination se fait également à travers un cadre théorique partant de la philosophie esthétique et de la littérature afin de définir mon utilisation du terme *esthétique*. D'un point de vue littéraire, je m'appuie sur les travaux de Gérard Genette et sa distinction entre fiction et diction (Genette, 1993), considérant l'esthétique du code comme sa diction, tandis que la poétique serait sa fiction. Entre littérature, philosophie et sciences cognitives se trouvent les oeuvres de Paul Ricoeur (Ricoeur, 2003) et de George Lakoff (Lakoff & Johnson, 1980), reliant composition verbale et évocation d'images mentales. Enfin, cette manifestation en surface est reprise par le philosophe de l'art Nelson Goodman dans son analyse des languages de l'art (Goodman, 1976), et notamment dans son exploration entre systèmes syntactiques et communication, ainsi que son approche scientifique des phénomènes artistiques.

3 Développement

Premièrement, il n'y a pas "un" code source, abstrait et désincarné, mais bien une multiplicité de codes sources, existant au sein de pratiques et de groups différents. Ces codes sources possèdent tous une possibilité de manifestation esthétique, manifestations soumise à la manifestation et au transfert de *connaissances*. En ce que le code source possède tant un son aspect prescriptif (ce que le code *doit* faire) qu'un aspect effectif (ce que

le code *fait*), les esthétiques du code source peuvent se décliner le long de cette axe, avec les poètes et les académiques se concentrant sur l'aspect prescriptif, évoquant des concepts à travers la machine, et les hackeurs se concentrant sur l'aspect effectif, évoquant les concepts de la machine.

Pour communiquer l'intention du programme à travers son implémentation textuelle et en rapport avec son domaine d'activité, les programmeurs disposent de différents possibilités. Il est possible de développer une certaine du problème vers à travers divers niveaux de métaphores linguistiques (rhétorique procédurelle (Bogost, 2008), double-codage (Cox & McLean, 2013), double-signification (Paloque-Bergès, 2009)) par un choix précis de vocabulaire. Le code est donc ici une double sorte de langue: langue machinique et langue humaine dont les tensions syntactiques peuvent créer une richesse sémantique.

Ensuite, ces mécanismes linguistiques sont complémentés par des choix de structures et de syntaxe faisant appel à des domaines d'architecture et d'ingénierie (Gabriel, 1998; Schummer et al., 2009), qui permettent alors de mieux appréhender une exécution du code source—exécution dont la vitesse à laquelle elle se produit la rend incompréhensible pour des humains. L'esthétique du code source permet alors de représenter les espaces d'évolution des flux d'informations lors de l'exécution d'un programme.

Finalement, ces standards esthétiques se regroupent autour de la notion d'élégance—c'est-à-dire de l'utilisation d'une syntaxe minimale pour une expressivité maximale. Particulièrement présente dans les groupes de hackeurs et des académiques, celle-ci se décline aussi le long d'un axe machine-concept. Les hackeurs ont tendance à n'utiliser que ce qui est nécéssaire pour effectuer une action particulière, bien visible dans les concours de *demoscenes* (Kudra, 2020), tandis que les académiques vont témoigner de la même approche, mais pour communiquer des concepts fondamentaux de la science informatique, découplée de la machine spécifique qui exécute le programme en question.

4 Idéaux discursifs

The history of software development is that of a practice born in the aftermath of the second world war, one which trickled down to broader and broader audiences at the eve of the twenty-first century. Through this development, various paradigms, platforms and applications have been involved in producing software, resulting in different epistemic communities (Cohendet et al., 2001) and communities of practice (Hayes, 2015), in turn producing different types of source code. Each of these write source code with particular characteristics, and with different priorities in how knowledge is produced, stored, exchanged, transmitted and retrieved. We delimitate these communities in four: software developers, hackers, scientists and poets. Taking a socio-historical stance on the field of programming, we highlight how diverse practices emerge at different moments in time, how they are connected to contemporary technical and economic organizations, and for specific purposes. Even though such types of reading and writing source code often overlap with one another, we highlight a diversity of ways in which code is written, notably in terms of origin—how did such a practice emerge?—, references—what do they consider good?—, purposes—what do they write for?—and examples—how does their code look like?.

Software developers operate in a commercial realm where the goals are to produce large, functional, sustainable software. They are closely tied to the information technologies industry, as well as to hardware and programming language development. Also starting in a university context, hackers write code that tends to be inscrutable, clever and very effective in the short term, participating in playful competitions such as code golf or obfuscation contests. Finally, software artists and code poets focus on source code as a textual and ambiguous semantic material, looking a compression in meaning by harnessing the intertwinings of natural and machine languages.

While none of these communities of practice are mutually exclusive—a software developer by day can hack on the weekend and participate in code poetry events—, they do help us grasp how source code's manifestations in program texts and its evaluation by programmers can be multifaceted. For instance, software engineers prefer code which is modular, modifiable, sustainable and understandable by the largest audience of possible contributors, while hackers would favor conciseness over expressivity, and tolerate playful idiosyncracy for the purpose of immediate, functional efficiency, with a practical engagement with the tools of their trade. On the other hand, scientific programming favors ease of use and reproducibility, along with a certain quest to represent the elegant concepts of computer science.

Still, there are strands of similarity within this apparent diversity. The code snippets in this section show that there is a tendency to prefer a specific group of qualities—readability, conciseness, clarity, expressivity and functionality—even though different types of practices would put a different emphasis on each of those aspects.

Following our overview of the varieties of practices and program texts amongst those who read and write source code, we now analyze more thoroughly what are the aesthetic standards most valued by those different groups. The aim here is to formalize our understanding of which source code is considered aesthetically pleasing. To do so, we capture both the specific manifestations of beautiful code enunciated by programmers, and identify the semantic contexts from which these enunciations originate, by using a discourse analysis framework for the empirical study of the corpus, followed by an examination of the discourses that programmers deploy when it comes to expliciting their aesthetic preferences of source code. What we will see is that, while the aesthetic domains that are mobilized to justify the aesthetic standards are clearly distinct, we can nonetheless identify recurrent sets of aesthetic values and a set of aesthetic manifestations against which the quality of source code can be measured.

Cleanliness is tied to expressiveness: by being devoid of any extraneous syntactic and semantic symbols, it facilitates the identification of the core of the problem at hand. Cleanliness thus works as a pre-requisite for expressivity. In a clean-looking program text, the extraneous details disappear at the syntactic level, in order to enable expressiveness at the semantic level. As a corollary to clarity stands obfuscation. It is the act, either intentional or un-intentional, to complicate the understanding of what a program does by leading the reader astray through a combination of syntactic techniques, a process we have already seen in the works of the IOCCC above. In its most widely applied sense, obfuscation is used for practical production purposes: reducing the size of code, and preventing the leak of proprietary information regarding how a system behaves. For instance, a program text can be obfuscated through a process called *minification*: the result is a shorter and lighter program text when it comes to its circulation over a network, at the expense of readability.

Simplicity in source code is therefore a form of parsimony and balance[1]. This requirement of exerting balance leads us to make a difference between two kinds of simplicity: syntactical simplicity, and ontological simplicity. Syntactic simplicity measures the number and conciseness of visible lexical tokens and keywords. Ontological simplicity, in turn, measures the number of kinds of entities involved in the semantics of the program text. Source code can have syntactic simplicity because it wrangles together complex concepts in limited amount of characters, or code can have ontological simplicity, because of the minimal amount of computational concepts involved. Syntactical simplicity also has a more immediate consequence on one's experience when reading a program text: one of the issues that programmers face is that there are just too many lines of code that one can wrap its head around, thus requiring that the content be pared

[1]Gibbons quotes Ralph Waldo Emerson to qualify his point: *"We ascribe beauty to that which is simple; which has no superfluous parts; which exactly answers its end; which stands related to all things; which is the mean of many extremes."* (Gibbons, 2012)

down to its functional minimum (Butler, 2012).

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

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

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

Cleverness is often found, and sometimes derided, in examples of code written by hackers, since it unsettles this balance between precision and generality. Clever code would tend towards exploiting particularities of knowledge of the medium (the code) rather than the goal (the problem).

Connected to simplicity by way of necessity and sufficiency, the perception of elegance is also related to a subjective feeling of adequacy, of fitness. Including some of the definitions of simplicity we have seen so far, Paul DiLascia, writing in the Microsoft Developer Network Magazine, illustrates his conception of elegance—as a combination of simplicity, efficiency and brilliance—with recursion (DiLascia, 2019), as seen in 1.

A complementary approach to understand what programmers mean when they talk about beautiful code is to look beyond the positive terms used to qualify it, and shift our attention to negative qualifiers. For instance, *spaghetti code* refers to a property of source code where the syntax

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

Listing 1: *factorical.c* - The use of recursion, rather than iteration, in the computation of a factorial is particularly praised by programmers.

is written in such a way that the order of reading and understanding is akin to disentangling a plate of spaghetti pasta. While technically still linear in execution, this linearity loses its cognitive benefits due to its extreme convolution, making it unclear what starts and ends where, both in the declaration and the execution of source code. Rather than using a synonym such as *convoluted*, the image evoked by spaghetti is particularly vivid on a sensual level, as a slimy, vaguely structured mass, even if the actual processes at play remain eminently formal (Steele, 1977). Such a material metaphor is used in a similar way in Foote and Yoder's description of code as a "big ball of mud"[2].

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

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

Moving from a syntactical level to a thematical level, to refer to Kintsch and Van Dijk's framework of discourse analysis, we now turn to an investigation of each of these domains, and what they tell us about source code.

The assumption here is that a medium—such as source code—is a means of expression, and different mediums can support different qualities of expression; additionally, a comparative analysis can be productive as it reveals the overlaps between these mediums. Since there seems to be some specific ways in which code can be considered beautiful, these adjacent domains, and the specific parts of these domains which create this contingency, will prepare our work of defining source code-specific aesthetic standards.

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

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

By referring to code as text, its linguistic nature is highlighted, along with its problematic relationship to natural languages—problematic insofar as its ambiguity can play against its desire to be clear and understood, or can play in favor of poetic undertones. The standards expressed here touch upon the specific syntax used to write programming, its relationship to natural language and its potential for expressivity.

Considering the formal nature of source code, scientific metaphors equate source code as having the potential to exhibit similar properties

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

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

Finally, beyond the references to those metaphors, we can also highlight the fact that aesthetics always seem to be connected to the need to *understand* software. We have highlighted a focus on understanding when it comes to aesthetic standards: whether obfuscating or illuminating, the process of acquiring a mental model of a given computational object is a key determinant in the value judgment as applied to source code.

5 Complexité des pratiques de programmation

Aesthetics in source code are thus primarily related to understanding. In this section, we focus on the reason for which software involves such a cognitive load, before surveying the means—both linguistic and mechanistic—that programmers deploy in order to relieve such a load.

This requirement for understanding, whether in a serious, playful or po-

etic manner, is related to one of the essential features of software: it must be *functional*. As mentioned in our discussion of the differences between source code and software in the introduction, source code is the latent description of what the software will ultimately do. Similarly to sheet music, or to cooking recipes[3], they require to be put into action in order for their users (musicians and cooks, respectively) to assess their value. Therefore, buggy or dysfunctional software is always going to be of less value than correct software (Hill, 2016), regardless of how aesthetically pleasing the source is. Any value judgment regarding the aesthetics of the source code would be subject to whether or not the software functions correctly, and such judgment is rendered moot if that software does not work.

The assessment of whether a piece of software functions correctly can be broken down in several sub-parts: knowing what the software effectively does, what it is supposed to do, being able to tell whether these two things are aligned, and understanding how it does it. After deciding on a benchmark to assess the functionality of the source code at hand (understanding what it should be doing), one must then determine the actual behavior of the source code at hand once it is executed (understanding what it is actually doing). Due to its writerly nature, one must also understand how a program text does it, in order to modify it.

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

However, the explicit specification of meaning falls short of handling

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

everyday tasks which humans would consider to be menial. This leads us to consider a different approach to understanding, in which it is acquired through contextual and embodied means (Penny, 2019). Particularly, we can identify tacit knowledge as relying on a social component, as well as on a somatic component (Polanyi & Grene, 1969).

Source code, as a formal system with a high dependence of context, intent and implementation, mobilizes both approaches to understanding. Due to programing's practical and bottom-up nature, attemps to formalize it have relied on the assumption that expert programmers have a certain kind of tacit knowledge (Soloway et al., 1982; Soloway & Ehrlich, 1984). The way in which this knowledge, which they are not able to verbalize, has been acquired and is being deployed, has long been an object of study in the field of software psychology.

Software involves, through programming languages, the expression of human-abstracted models for machine interpretation, which in turn is executed at a scale of time and space that are difficult to grap for individuals. These properties make it difficult to understand, from conception to application: software in the real-world goes through a process of implementation of concepts that lose in translation, interfacing the world through discrete representations, and following the execution of these representations through space and time. In this process, source code remains the material representation of all of these dynamics and the only point of contact between the programmer's agency and the machine execution and, as such, remains the locus of understanding.

Metaphors play a role in understanding in creating connections between pre-existing knowledge and current knowledge, suggesting similarities between both in order to facilitate the construction of mental models of the target domain. Metaphors are used by programmers at a different level, helping them grasp concepts (e.g. memory, objects, package) without having to bother with details.

Programmers also rely on specific software tools, in order to facilitate

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

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

In the next section, we pursue our inquiry of the means of understanding, moving away from software, and focusing on how the aesthetic domains. This will allow us to show how source code aesthetics, as highlighted by the metaphorical domains that refer to it, have the function of making the imperceptible understandable.

6 Esthétique et cognition

Jack Goody and Walter Ong have shown in their anthropological studies that the primary means of communication of the surveyed communities does affect the engagement of said communities with concepts such as ownership, history and governance (Ong, 2012; Goody, 1986). More recently, Edward Tufte and his work on data visualization have furthered this line of research by focusing on the translation of similar data from textual medium to graphic medium (Tufte, 2001). Several cases have thus been made for the impact of appearance towards structure, both in source

code and elsewhere. Here, we intend to generalize this comparative approach between several mediums, by looking at how source code performs expressively as a language of art, stemming from Nelson Goodman's theorization of such a languages.

A symbol system is based on requirements which might indicate that the work created in such a system would be able to elicit an aesthetic experience[4]. Such a system should be composed of signs which are syntactically and semantically disjointed, syntactically replete and semantically dense (Goodman, 1976). This classification makes it possible to compare the way various symbolization systems used in art and sience express concepts. In our case, this provides us for a framework to investigate the extent to which source code qualifies as a language of art.

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

Beardsley highlights the possibility of an aesthetic experience to make understandable, through *active discovery*, and he considers metaphors as a way to do so (Beardsley, 1970). The stages he lists go from (1) the word exhibiting properties, to (2) those properties being made into meaning, and

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

finally into (3) a staple of the object, consolidating into (or dying from becoming) a commonplace. This interplay of a metaphor being integrated into our everyday mental structures, of poetry bringing forth into the thinkable, and in the creation of a tension for such bringing-forth to happen, makes the case for at least one of the consequences of an aesthetic experience, and therefore one of its functions: making sense of the complex concepts of world.

Catherine Elgin has pursued the work of Goodman by furthering the inquiry into arts as a branch of epistemology. Drawing on the work mentioned above, she investigates the relationship between art and understanding, considering how interpretively indeterminate symbols advance understanding (Elgin, 2020), and that it does so in the context of interpretive indeterminacy. As syntactically and semantically dense symbol systems are used in artworks, it is this multiplicity in interpretations which requires sustained cognitive attention with the artwork. To explain these multiple interpretations, the metaphor is again presented the key device in explaining the epistemic potency of aesthetics, based on an interpretative feedback loop from the viewer. And yet, in the context of source code, this interpretation is always shadowed by its machine counterpart—how the computer interprets the program.

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

While these other types of experiences remain valid when apprehending such an object, we do focus here on this specific kind of experience:

the cognitive approach to the aesthetic experience[5]

Syntactical literary devices allow readers to engage cognitively with a particular content; they enable the construction of mental models a particular narrative, through a network of metaphors, allusions, ambiguous interpretations and chronotopes (Bakhtin, 1981). These literary devices also apply to source code, especially how the use of machine tokens and human interpretation suggest an aesthetic experience through metaphors, and with particular markers that are needed to make sense of the time and space of a computer program, which differs radically from that of a printed text. This making sense of a foreign time and space is indeed essential in creating a mental map of the storyworld (in fiction) or the world of reference (in non-fiction) (Ryan, 2009).

The use of the term map also implies a specific kind of territory, enabled by the digital. As a hybrid between the print's flatness and code's depth, Ryan and Murray—among many others—identify the digital narrative as a highly spatialized one (Murray, 1998). This feature, Ryan argues, is but a reflection of the inner architecture of source code. Pushing this line of thought further, we now turn to architecture as a discipline to investigate how the built environment elicits understanding, and how such possibilities might translate in the space of program texts.

Architecture can also offer some heuristics when looking for aesthetic features which code can exhibit. Starting from the naïve understanding that form should follow function, we've examined how Alexander's theory of patterns (Alexander et al., 1977), and its significant influence on the programming community[6], points not just to an explicit conditioning of form to its function (in which case we would all write hand-made Assembly

[5]Going back to Goodman, he describes such an experience as involving:*making delicated discriminations and discerning subtle relationships, identifying symbol systems and what these characters denote and exemplify, interpreting works and reorganizing the world in terms of works of art and works in terms of the world.*.(Goodman, 1976)

[6]This theory has even spawned short-lived debates about his quality without a name on stackoverflow (interstar, 2017).

code), but rather to an elusive, yet present quality, which is both problem- and context-dependent.

Along with the function of the program as an essential component of aesthetic judgment, our inquiry has also shown that program texts can present a quality that is aware of the context that the writer and reader bring with them, and of the context that it provides them, making it habitable. Software architecture and patterns are not, however, explicitly praised for their beauty, perhaps because they disregard these contexts, since they are higher-level abstractions; this implies that generic solutions are rarely elegant solutions. And yet, there is an undeniable connection between the beautiful, the crafted and the universal.

Aesthetics are closely involved in considering mathematical objects, in appreciating their symbolic representation (Hardy, 2012; Poincaré, 1908), and as a heuristic towards a positively-valued representation (Sinclair, 2004). Particularly, the dichotomy between the mathematical entities (theorems) and their representations (proofs) echoes the distinction we have seen in programming between algorithm and implementation. While abstract entities do possess specific qualities that are positively valued, it is their implementation—that is, their textual manifestation—which tends to be the locus of aesthetic judgment. Aesthetics also complement the more common notion of scientific rational thinking, in which an individual reasons about a problem in a linear, step-by-step manner. Instead, we have seen that the appearance, and the judgment of such appearance, also acts as a guide towards establishing true and elegant mathematical statements.

Ultimately, aesthetics in mathematics contribute to representing a mathematical object, thus enabling access to the conceptual nature and implications of this object, as well as providing useful heuristics in establishing a new object. What remains, and what will be taken up in the next chapter, is to *"reify this meta-logic as a set of rules, axioms, or practices."* (Root-Bernstein, 2002), by establishing which mathematical approaches fit

with source code aesthetics.

7 Esthétique spatiale des codes sources

Software is thus the representation of an idea into specific hardware configurations. The immediate medium of this representation, from the programmer's perspective, is the programming language in which the idea is written down. Programming languages have so far been set aside when examining which sensual aspects of source code resulted in what could be deemed a "beautiful" program text. And yet, the relationship between semantics (deep-structure) and its syntactic representation (surface-structure) is framed by programming languages, as they define the legal organization of form.

Programming languages are both tools and environments, and moreover eminently *symbolic*, manipulating and shaping *symbolic* matter. Looking at these languages from a Goodmanian perspective provides a backdrop to examine their communicative and expressive power. From the perspective of the computer, programming languages are unambiguous insofar as any expression or statement will ultimately result in an unambiguous execution by the CPU. They are also syntactically disjointed, but not semantically: two programming tokens can have the same effect under different appearances. The use of formal specifications aims at resolving any possible ambiguity in the syntax of the language in a very clear fashion, but fashionable equivalence can come back as a desire of the language designer. The semantics of programming languages, as we will see below, also aim at being somewhat disjointed: a variable cannot be of multiple types at the exact same time, even though a function might have multiple signatures in some languages. Finally, programming languages are also differentiated systems since no symbol can refer to two things at the same time.

The tension arises when it comes to the criteria of unambiguity, from a human perspective. The most natural-language-like component of programs, the variable and function names, always have the potential of being ambiguous[7]. We consider this ambiguity both a productive opportunity for creativity, and a hindrance for program reliability. If programming languages are aesthetic symbol systems, then they can allow for expressiveness, first and foremost of computational concepts. It is in the handling of particularly complex concepts that programming languages also differentiate themselves in value. The differences in programming language design and us thus amounts to differences in style[8].

Concrete use of programming languages operate on a different level of formality: if programming paradigms are top-down strategies specified by the language designers, they are also complemented by the bottom-up tactics of softare developers. Such practices crystallize, for instance, in *idiomatic writing*. Idiomaticity refers, in traditional linguistics, to the realized way in which a given language is used, in contrast with its possible, syntactically-correct and semantically-equivalent, alternatives. For instance, it is idiomatic to say "The hungry dog" in English, but not "The hungered dog" (a correct sentence, whose equivalent is idiomatic in French and German). It therefore refers to the way in which a language is a social, experiential construct, relying on intersubjective communication (Voloshinov & Bachtin, 1986). Idiomaticity is therefore not a purely theoretical feature, but first and foremost a social one. This social component in programming languages is therefore related to how one writes a language "properly".

[7] For instance, does `int numberOfFlowers` refer to the current number of flowers in memory? To the total number of potential of flowers? To a specific kind of number whose denomination is that of a flower?

[8] In the words of Niklaus Wirth: *"Stylistic arguments may appear to many as irrelevant in a technical environment, because they seem to be merely a matter of taste. I oppose this view, and on the contrary claim that stylistic elements are the most visible parts of a language. They mirror the mind and spirit of the designer very directly, and they are reflected in every program written.* (Wirth, 2003)"

In this sense, programming language communities are akin to hobbyists clubs, with their names[9] meetups, mascots, conferences and inside-jokes. Writing in a particular language can be due to external requirements, but also to personal preference [10].

So an idiom in a programming language depends on the social interpretation of the formal programming paradigms[11]. Such an interpretation is also manifested in community-created and community-owned documents.

We have seen how programming languages can be subjected to aesthetic judgment, but those aesthetic criteria are only there to ultimately support the writing of good (i.e. functional and beautiful) code. Such a support exists via design choices (abstraction, orthogonality, simplicity), but also through the practical uses of programming languages, notably in terms of idiomaticity and of syntactic sugar, allowing some languages more readability than others. Like all tools, it is their (knowledgeable) use which matters, rather than their design, and it is the problems that they are used to deal with, and the way in which they are dealt with which ultimately informs whether or not a program text in that language will exhibit aesthetic features.

This concept of appropriateness also relates to material honesty. As seen in the fact that a programmer tends to identify their practice with craft implies that they work with tools and materials. Programming languages being their tools, and computation the material, one can extend to the concept of material honesty to the source code (Sennett, 2009). In

[9] Pythonistas for Python, Rubyists for Ruby, Rustaceans for Rust, Gophers for Go, etc.

[10] *"I think a programming language should have a philosophy of helping our thinking, and so Ruby's focus is on productivity and the joy of programming. Other programming languages, for example, focus instead on simplicity, performance, or something like that. Each programming language has a different philosophy and design. If you feel comfortable with Ruby's philosophy, that means Ruby is your language."* (Matsumoto, 2019)

[11] This is even more present in contemporary programming languages, since paradigms in these languages are often blended and no language is purely single-paradigmatic; for instance, Ruby is a declarative language with functional properties (Kidd, 2005)

this case, working with, and in respect of, the material and tools at hand is a display of excellence in the community of practitioners, and results in an artefact which is in harmony and is well-adapted to the technical environment which allowed it to be. Source code written in accordance with the principles and the affordances of its programming language is therefore more prone to receive a positive aesthetic judgment. Furthermore, idiomatic writing is accompanied by a language-independent, but group-dependent feature: that of programming style.

The discourses of programmers in our corpus do not contain unidimensional criteria, but rather criteria which can be applied at multiple levels of reading. Some tend to relate more to the over-arching design of the code examined while others focus on the specific formal features exhibited by a a given token or successions of tokens in a source code snippet. To address this variability of focus, we borrow from John Cayley's distinction between structures, syntaxes and vocabularies (Cayley, 2012). Cayley's framework will allow us to take into account an essential aspect of source code: that of scales at which aesthetic judgment operates. Beyond literary studies, this framework is also used by Dijkstra when he introduces his approach to structured programming, from the high-level of the program taken as a whole down to the details of line-by-line syntactic choices (Dijkstra, 2007). From the psychological accounts of understanding source code in to the uses of space in domain-specific aesthetics in or one of the specificities of source is the multiple dimensions of its deep structure hidden behind the two-dimensional layout of a text file, and the need for programmers to navigate such space.

Structure is defined by the relative location of a particular statement within the broader context of the program text, as well as the groupings of particular statements in relation to each other and in relation to the other groups of statements within the program-text, whether it is across the same file, series of files, or a sprawling network of folders and files. This also includes questions of formatting, indenting and linting as purely

pattern-based formal arrangements, as seen in since these affect the whole of the program-text.

Syntax concerns the local arrangement of tokens within a statement or a block statement, including control flow, iterators statements, function declarations, etc., which can be referred to as the "building blocks" of a program-text. Syntax also includes language-specific choices—idioms—and generally the type of statements needed to best represent the task required (e.g. using an array or a struct as a particular data structure for representing entities from the problem domain).

Finally, the vocabulary refers to the user-defined elements of the source code, in the form of variable, function, classe and interface names. Source code vocabulary is constituted of both reserved keywords (which the computer "understands" by being explicitly mentioned by the language designers) and user-defined keywords, the single words which the writes defines themselves and which are not known to a reader who would already know the language's keywords. Unlike the two precedent categories, this is therefore the only one where the writer can come up with new tokens, and is the closest to metaphors in traditional literature.

The specificity of source code is that it acts as a techno-linguistic interface between two meaning-makers: the human and the machine. While the machine has a very precise, operational definition of meaning, programmers tend to mobilise different modalities in order to make sense of the system they are presented with through this textual interface. Among those modalities are the resort to literary techniques (in the form of metaphors), to architecture (in the form of pattern-based structural organization), to mathematics (in the form of symbolic elegance) and craft (in the form of material adequacy and reliabilty).

As a formal manifestation involving, in the context of a crafted object, a producer and a receiver, aesthetics contribute to the establishment of mental spaces. In domains such as mathematics or literature, mental spaces can represent theorems or emotions; within source code, however, they ac-

quire a more functional dimension. As such, they also communicate *states* and *processes*.

Building on our discussion of understanding software in we now highlight concrete instances of complex computational objects interfaced through source code. We take three examples to highlight the multiple manifestations of semantic layers at play in source code, operating in different socio-technical contexts, yet all sharing the same properties of using structure, syntax and vocabulary in order to communicate implicitly a relatively complex idea. We can consider a semantic layer to be an abstraction over relatively disorganized data in order to render it relatively organized, by providing specific reference points which contribute to the establishment of a mental space, based on the computational space of the program[12].

The expressivity of program texts rely on several aesthetic mechanisms, connected in a spatial way between a metaphorical understanding of humans and a functional understanding of machines. From layout to double-meaning through variables and procedure names (Paloque-Bergès, 2009), double-coding and the integration of data types and functional code into a program text (Cox & McLean, 2013) and a rhetoric of procedures in their written form, all of these activate the connection between programming concepts and human concepts to bring the unthinkable within the reach of the thinkable. While these techniques are deployed differently according to the socio-technical environment in which the program text is being written and read, they nonetheless all contribute to faciltating the navigation of the program text, be it at the same level of abstraction across parts of the text, or at different levels of abstraction in the same locations where the syntax abstracts away the unnecessary signifiers of

[12] *" A challenge in computer system design is that the representation of the functionality at any particular layer of abstraction should exhibit just those characteristics that are essential at that layer, without the clutter of notational obfuscation and unnecessary appearance of underlying complexity."* (Neumann, 1990)

parallel computing.

Ultimately, these aesthetic manifestations of source code in a program text are all tightly coupled to the execution of that program text. Through our comparative study of source code with architecture and mathematics, we have seen that aesthetics are not unrelated to ideas of function, or purpose. In the case of architecture, an aeshetic appreciation of a building can hardly be made completely independently from the building's intended function while, in the case of mathematics, aesthetics are closely associated with an epistemological function. In turn, the aeshetics of source code have been shown to be closely connected to different kinds of cognitive engagement, from clarity to obfuscation and metaphorical evokation.

Each program text that we have examined in this work always implies a necessity of being functioning in order to be properly judged at an aesthetic level, but the diversity of practices we have pointed to also seems to suggest different conception of functions. In developing the different ways that the function of a program text can be considered, we argue for a dual relation ship between function and aesthetics in source code. First, the function of a program text is integral in the aesthetic judgment of such text, both because the status of software as crafted object makes its function the deep structure that is manifested in its surface, and because the standards by which it is judged depend on socio-technical contexts in which the program text is meant to be used. Second, the aesthetics of source code are not autotelic; rather, the function of aesthetics themselves is to communicate invisible information to the reader and writer.

All source code aesthetics relate to a certain conception of function, involving technical achievement and interpersonal existence. An aesthetic judgment of a program text is, in this understanding, the judgment of the perceptible manifestations in source codes allowing for the comprehension of a technical achievement according to contextual standards. These manifestations are therefore not just expressive (personal), but primarily communicative (interpersonal), aiming at the transmission of concepts

from one individual through the use of machine syntax through the dual lens of human-machine semantics. Indeed, code that is neither functioning for the machine, nor meaningful for a human holds the least possible value amongst practitioners.

In the overwhelming majority of cases of program texts, the expectation is to understand. Writing aesthetically pleasing code is to write code that engages cognitively its reader, whether explicitly for software developers, pedagogically for scientists, adversely for hackers and metaphorically for poets. This engagement, in turn, supposes an acknowledgement from the writer of the reader. The recognition of the existence of the other as a reader and co-author, implies an acknowledgement as a generalized other in the sense that anyone can theoretically read and modified code, but also as a specificied other, in the sense that the other possesses a particular set of skills, knowledge, habits and practices stemming from the diversity of programming communities. This stance, between general and particular, is one that shows the ethical component of an aesthetic practice: recognizing both the similarity and the difference in the other, and communicating with a peer through specific symbol systems.

8 Conclusion

Programming languages can thus be considered materially, as *the interplay between a text's physical characteristics and its signifying strategies* (Hayles, 2004), which in turn depend on socio-technical dynamics. As an interface to the computer, programming languages, without overly-determining the practice of programmers or the content of what is being programming, programming languages nonetheless influence how it can be said, through idiosyncracies and stylistic devices. This has established the idiosyncratic status of source code as a medium, and its existence between technical and social, expressive and communicative, individual and collaborative.

We then presented a framework for aesthetics of source code, through the dual lense of semantic compression and spatial navigation. To do so, we started from a layer-based approach to the points in which aesthetic decisions can take place in source code—that is, across structure, syntax and vocabulary. Broadening this approach, we then showed how these different levels involve an engagement with semantic layers: between the human reader, the machine reader and the problem domain. The minimizing of syntax while best representing the different concepts involved at these different layers results in semantic compression. A source code with aesthetic value is one which balances syntactic techniques, structural organization and metaphorical choices in order to communicate a socio-technical intent of a functional artefact. In turn, semantic compression supports the shifts from different scales or perspectives the engaged programmer needs to operate as she navigates through her non-linear exploration of a program text.

En fin de compte, nous avons montré que les représentations métaphoriques du code, représentations du code comme language, comme architecture, et comme matériau, se retrouvent dans l'expression d'une conception d'un espace sémantique dynamique. Cette conception spatiale infuse donc les propriétés esthétiques propres du code, en ce que la syntaxe et la structure du code source se concentrent principalement à la description et à la navigation d'espaces conceptuels, même si les différents contextes socio-techniques dans lesquels ces codes sources sont écrits vont eux-mêmes moduler la nature de ces espaces.

References

Abelson, H., Sussman, G. J., & Sussman, J. (1979). *Structure and Interpretation of Computer Programs - 2nd Edition*. Justin Kelly.

Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., & Angel, S. (1977). *A Pattern Language: Towns, Buildings, Construction*. New York: Oxford University Press.

Bakhtin, M. M. M. M. (1981). *The Dialogic Imagination : Four Essays*. Austin : University of Texas Press.

Beardsley, M. C. (1970). The Aesthetic Point of View. *Metaphilosophy*, *1*(1), 39–58.

Black, M. J. (2002). The art of code. *Dissertations available from ProQuest*, (pp. 1–228).

Bogost, I. (2008). The Rhetoric of Video Games. In K. Salen (Ed.) *The Ecology of Games: Connecting Youth, Games and Learning*. Cambridge, MA: The MIT Press.

Butler, B. (2012). On Programmer Archaeology.

Cayley, J. (2012). The Code is not the Text (Unless It Is the Text) › electronic book review.

Cohendet, P., Creplet, F., & Dupouët, O. (2001). Organisational innovation, communities of practice and epistemic communities: The case of linux. In A. Kirman, & J.-B. Zimmermann (Eds.) *Economics with Heterogeneous Interacting Agents*, (pp. 303–326). Berlin, Heidelberg: Springer Berlin Heidelberg.

Cox, A., & Fisher, M. (2009). Programming Style: Influences, Factors, and Elements. In *2009 Second International Conferences on Advances in Computer-Human Interactions*, (pp. 82–89).

Cox, G., & McLean, C. A. (2013). *Speaking Code: Coding as Aesthetic and Political Expression*. MIT Press.

Cramer, F. (2019). *Exe.cut(up)able statements: Poetische Kalküle und Phantasmen des selbstausführenden Texts*. Wilhelm Fink.

Detienne, F. (2001). *Software Design – Cognitive Aspect*. Springer Science & Business Media.

Dijkstra, E. W. (2007). The humble programmer. In *ACM Turing Award Lectures*, (p. 1972). New York, NY, USA: Association for Computing Machinery.

DiLascia, P. (2019). { End Bracket }: What Makes Good Code Good? URL `https://docs.microsoft.com/en-us/archive/msdn-magazine/2004/july/\%7b-end-bracket-\%7d-what-makes-good-code-good`

Elgin, C. Z. (2020). Understanding Understanding Art. In V. Granata, R. Pouivet, & Université de Lorraine (Eds.) *Épistémologie de l'esthétique: perspectives et débats*, Collection "Hors-série". Rennes: Presses universitaires de Rennes.

Foote, B., & Yoder, J. (1997). Big Ball of Mud. URL `http://www.laputan.org/mud/mud.html\#BigBallOfMud`

Gabriel, R. P. (1998). *Patterns of Software: Tales from the Software Community*. Oxford University Press.

Gabriel, R. P., & Goldman, R. (2001). *Mob Software: The Erotic Life of Code*.

Genette, G. (1993). *Fiction & Diction*. Cornell University Press.

Gibbons, J. (2012). The beauty of simplicity. *Communications of the ACM*, *55*(4), 6–7.

Goodman, N. (1976). *Languages of Art*. Indianapolis, Ind.: Hackett Publishing Company, Inc., 2nd edition ed.

Goody, J. (1986). *The Logic of Writing and the Organization of Society*. Studies in Literacy, the Family, Culture and the State. Cambridge University Press.

Graham, P. (2003). Hackers and Painters.
URL http://www.paulgraham.com/hp.html

Hardy, G. H. (2012). *A Mathematician's Apology*. Canto Classics. Cambridge: Cambridge University Press.

Hayes, B. (2015). Cultures of Code. *American Scientist*, *103*(1).

Hayles, N. K. (2004). Print Is Flat, Code Is Deep: The Importance of Media-Specific Analysis. *Poetics Today*, *25*(1), 67–90.

Hayles, N. K. (2010). *My Mother Was a Computer: Digital Subjects and Literary Texts*. University of Chicago Press.

Hill, R. K. (2016). What Makes a Program Elegant?
URL https://cacm.acm.org/blogs/blog-cacm/208547-what-makes-a-program-elegant/fulltext

interstar (2017). Quality Without A Name (QWAN) examples?

Kidd, E. (2005). Why Ruby is an acceptable LISP (2005) | Random Hacks.
URL http://www.randomhacks.net/2005/12/03/why-ruby-is-an-acceptable-lisp/

Kintsch, W., & van Dijk, T. A. (1978). Toward a model of text comprehension and production. *Psychological Review*, *85*(5), 363–394.

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

Kudra, A. (2020). AoC \textbackslashtextbackslashtextbar Art of Coding – The Demoscene as Intangible World Cultural Heritage. In C. Yackel, R. Bosch, E. Torrence, & K. Fenyvesi (Eds.) *Proceedings of Bridges 2020: Mathematics, Art, Music, Architecture, Education, Culture*, (pp. 479–480). Phoenix, Arizona: Tessellations Publishing.

Lakoff, G., & Johnson, M. (1980). *Metaphors We Live By*. University of Chicago Press.

Lévy, P. (1992). *De la programmation considérée comme un des beaux-arts*. Textes à l'appui. Anthropologie des sciences et des techniques. Paris: Éd. la Découverte.

Mackenzie, A. (2006). *Cutting Code: Software and Sociality*. Peter Lang.

Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Pearson Education.

Matsumoto, Y. (2019). Yukihiro Matsumoto: "Ruby is designed for humans, not machines".
URL `https://evrone.com/blog/yukihiro-matsumoto-interview`

McLean, A. (2004). Hacking Perl in Nightclubs.
URL `https://perl.com/pub/2004/08/31/livecode.html/`

Montfort, N., Baudoin, P., Bell, J., Bogost, I., & Douglass, J. (2014). *10 PRINT CHR$(205.5+RND(1)); : GOTO 10*. The MIT Press, illustrated edition ed.

Murray, J. H. (1998). *Hamlet on the Holodeck: The Future of Narrative in Cyberspace*. Cambridge, MA, USA: MIT Press.

Neumann, P. G. (1990). Beauty and the Beast of Software Complexity — Elegance versus Elephants. In W. H. J. Feijen, A. J. M. van Gasteren, D. Gries, & J. Misra (Eds.) *Beauty Is Our Business: A Birthday Salute to Edsger W. Dijkstra*, Texts and Monographs in Computer Science, (pp. 346–351). New York, NY: Springer.

Ong, W. J. (2012). *Orality and Literacy: 30th Anniversary Edition*. London: Routledge, 3 ed.

Oram, A., & Wilson, G. (Eds.) (2007). *Beautiful Code: Leading Programmers Explain How They Think*. Beijing ; Sebastapol, Calif: O'Reilly Media, 1st edition ed.

Overflow, S. (2013). How can you explain "beautiful code" to a non-programmer?

Paloque-Bergès, C. (2009). *Poétique des codes sur le réseau informatique*. Archives contemporaines.

Penny, S. (2019). *Making Sense: Cognition, Computing, Art and Embodiment*. MIT Press.

Pineiro, E. (2003). *The Aesthetics of Code : On Excellence in Instrumental Action*. Ph.D. thesis, KTH, Superseded Departments, Industrial Economics and Management.

Poincaré, H. (1908). *Science et méthode*. Paris: E. Flammarion.

Polanyi, M., & Grene, M. (1969). *Knowing and Being; Essays*. [Chicago] University of Chicago Press.

Ricoeur, P. (2003). *The Rule of Metaphor: The Creation of Meaning in Language*. Psychology Press.

Root-Bernstein, R. S. (2002). Aesthetic cognition. *International Studies in the Philosophy of Science*, *16*(1), 61–77.

Ryan, M.-L. (2009). Space. In *Space*, (pp. 420–433). De Gruyter.

Schummer, J., MacLennan, B., & Taylor, N. (2009). Aesthetic Values in Technology and Engineering Design. In A. Meijers (Ed.) *Philosophy of Technology and Engineering Sciences*, Handbook of the Philosophy of Science, (pp. 1031–1068). Amsterdam: North-Holland.

Sennett, R. (2009). *The Craftsman*. Yale University Press.

Sinclair, N. (2004). The Roles of the Aesthetic in Mathematical Inquiry. *Mathematical Thinking and Learning*, *6*(3), 261–284.

Soloway, E., & Ehrlich, K. (1984). Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering*, *SE-10*(5), 595–609.

Soloway, E., Ehrlich, K., & Bonar, J. (1982). Tapping into tacit programming knowledge. In *Proceedings of the 1982 Conference on Human Factors in Computing Systems*, (pp. 52–57).

Steele, G. L. (1977). Macaroni is better than spaghetti. *ACM SIGPLAN Notices*, *12*(8), 60–66.

Tufte, E. R. (2001). *The Visual Display of Quantitative Information*. Graphics Press.

Voloshinov, V. N., & Bachtin, M. M. (1986). *Marxism and the Philosophy of Language*. Harvard University Press.

Weinberg, G. M. (1998). *The Psychology of Computer Programming*. Dorset House Pub.

Wirth, N. (2003). The Essence of Programming Languages. In L. Böszörményi, & P. Schojer (Eds.) *Modular Programming Languages*, Lecture Notes in Computer Science, (pp. 1–11). Berlin, Heidelberg: Springer.

Zeller, J. (2020). Algorithms are like recipes.
URL `https://www.goethe.de/en/kul/ges/21877729.html`

Zenil, H. (2021). Compression is Comprehension, and the Unreasonable Effectiveness of Digital Computation in the Natural World.