The role of aesthetics in the understandings of source code

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

ED120 - THALIM

last updated: 2023-08-02

Chapter 3

Understanding source code

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

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

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

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

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

This chapter examines what goes into understanding source code: given a certain nature of knowledge acquisition, we look at some of the features of computers that make them hard to grasp, and the kind of techniques are deployed in order to address these hurdles. This will have us investigate the relationship of knowing and doing, the nature of computation (what is software?) and its relationship to the world as it appears to us (how does modelling and abstraction translate a problem domain into software?), and the cognitive scaffoldings set up in response to facilitate that task. Ultimately, we show that, given our definition of understanding, the complex nature of software objects and the diverse techniques programmers use to grasp these objects, aesthetics of source code also hold a significant place in this understanding process, a position we develop in 5.

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

Having highlighted this tension, we then turn to how understanding

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

Finally, we will conclude with the means that programmers deploy to grasp the concepts at play with software: starting from metaphors used by the general public, we will then see to what extent they differ from the metaphors used by programmers in order to understand the systems they build and work with. In the end, particular attention will be paid to their extended cognition the technical apparatuses used in the development and inspection of source code.

3.1 Formal and contextual understandings

This section elaborates our definition of understanding—the process of acquiring a working knowledge of an object[2]. Such definition relies on two main aspects: a formal, abstract understanding, and a more subjective, empirical one. We will see how the former had some traction in computer sciences circles, while the second gained traction in programming circles. To support those two approaches, we first trace back the genealogy of understanding in theoretical computer science, before outlining how concrete complementary approaches centered around experience and situatedness

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

outline an alternative tradition.

3.1.1 Between formal and informal

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

Theoretical foundations of formal understanding

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

Wittgenstein, in particular, bases his argumentation in his *Tractatus Logico-philosophicus* on the fact that much of the problems in philosophy are rather problems of understanding between philosophers—if one were to express oneself clearly, and to articulate one's through clear, unambiguous language, a common conclusion could be reached without much effort[3]. The stakes presented are thus those of understanding what language really is, and how to use it effectively to, in turn, make oneself understood.

The demonstration that Wittgenstein undertakes is that language and logic are closely connected. Articulated in separate points and sub-points, his work conjugates aphorisms with logical propositions depending on one another, developing from broader statements into more specific pre-

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

cisions, going down levels of abstraction through increasing bulleted lists. Through the stylistic organization of his work, Wittgenstein hints at the possibility to consider language, itself pre-requisite for understanding, as a form of logic This complements the older approach to consider logic as a form of language. In this sense, he stands in the lineage of Gottfried Leibniz's *Ars Combinatoria*, since Leibniz considers that one can formalize a certain language (not necessarily natural languages such as German or Latin), in order to design a perfectly explicity linguistic system. A universal, and universally-understandable language, called a *characteristica universalis* could resolve any misunderstanding issues. Quoted by Russell, Leibniz notes that:

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

Centuries after Leibniz's declaration, Wittgenstein presents a coherent, articulated theory of meaning through the use of mathematical philosophy, and logic. His work also fits with that of Bertrand Russell and Alfred Whitehead who, in his *Principia Mathematica*, attempt to lay out a precise and convenient notation in order to express mathematical notations; similarly, Gottlieb Frege's work attempted to constitute a language in which all scientific statements could be evaluated, by paying particular attention to clarifying the semantic uncertainties between a specific sentence and how it refers to a concept (Korte, 2010).

Even though these approaches differ from, and sometimes argue with[4], one another, we consider them to be part of a broad endeavour to find a

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

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

Such works on formal languages as a means of knowledge processing a direct influence in the work on mathematician Alan Turing—who studied at Cambridge and followed some of Wittgenstein's lectures—, as he developed his own formal system for solving complex, abstract mathematical problems, manifested as a symbolic machine (Turing, 1936). Meaning formally expressed was to be mechanically processed.

The design of this symbol-processing machine, subsequently known as the Turing machine, is a further step in engaging with the question of knowledge processing in the mathematical sense, as well as in the practical sense—a formal proof to the *Entscheidungsproblem* solved mechanically. Indeed, it is a response to the questions of translation (of a problem) and of implementation (of a solution), hitherto considered a basis for understanding, since solving a mathematical problem supposed, at the time, to be able to understand it.

This formal approach to instructing machines to operate on logic statements then prompted Turing to investigate the question of intelligence and comprehension in *Computing Machinery and Intelligence*. In it, he translates the hazy term of "thinking" machines into that of "conversing" machines, conversation being a practical human activity which involves listening, understanding and answering (i.e. input, process and output; or attention, comprehension, diction) (Turing, 2009). This conversational test, which has become a benchmark for machine intelligence, would naively imply the need for a machine to *understand* what is being said.

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

---

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

would be provided as an input to a digital computer. Such a process of translation would rely on a formal approach, similar to that laid out in the *Tractatus Logico-philosophicus*, or on Frege's formal language described in the *Begriffschrift*. Following a cartesian approach, the idea in both authors is to break down a concept, or a proposition, into sub-propositions, in order to recursively establish the truth of each of these sub-propositions, and then re-assembled to deduce the truth-value of the original proposition.

Logical calculus, as the integration of the symbol into relationships of many symbols formally takes place through two stylistic mechanisms, the *symbol* and the *list*. Each of the works by Frege, Russell and Wittgenstein quoted above are structured in terms of lists and sub-lists, representing the stylistic pendant to the epistemological approach of related, atomistic propositions and sub-propositions. A list, far from being an innate way of organizing information in humans, is a particular approach to language: extracting elements from their original, situated existence, and reconnecting ways in very rigorous, strictly-defined ways[5].

As inventories, early textbooks, administrative documents as public mnemotechnique, the list is a way of taking symbols, pictorial language elements in order to re-assemble them to reconstitute the world, then re-assemble it from blocks, following an assumption that the world can always be decomposed into smaller, discreete and *conceptually coherent* units (i.e. symbols). One can then decompose a thought in a list, and expect a counterpart to recompose this thought by perusing it. As a symbol system, lists establish clear-cut boundaries, are simple, abstract and discontinuous; incidentally, this makes it very suited to a discreete symbol-processing machine such as the computer (Depaz, 2023).

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

[5]Jack Goody develops the influence of notation on cognition: *"[List-making] [...] is an example of the kind of decontextualization that writing promotes, and one that gives the mind a special kind of lever on 'reality'."* (Goody, 1977)

computer to achieve the intellectual capacities of a human brain.

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

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

Practical attempts at implementing formal understanding

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

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

IPL was originally designed to demonstrate the theorems of Russell's *Principia Mathematica*, along with a couple of early AI programs, such as the *Logic Theorist*, the *General Problem Solver*. As such, it proves to be a

link between the ideas exposed in the writing of the mathematical logicians and the actual design and construction of electrical machines activating these ideas. More a proof of concept than a versatile language, IPL was then quickly replaced by LISP as the linguistic means to express intelligence in digital computers (see 2.1.3).

This structure of Lisp is quite similar to the approach suggested by Noam Chomsky in his *Syntactic Structures*, where he posits the tree structure of language, as a decomposition of sentences until the smallest conceptually coherent parts (e.g. Phrase -> Noun-Phrase + Verb-Phrase -> Article + Substantive + Verb-Phrase). The style is similar, insofar as it proposes a general ruleset (or the at least the existence of one) in order to construct complex structures through simple parts.

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

> *There are fundamental gulfs between the way that SHRDLU and its kin operate, and whatever it is that goes on in our brains.*

I don't think that current research has made much progress in crossing that gulf, and the relevant science may take decades or more to get to the point where the initial ambitions become realistic. (Nilsson, 2009)

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

A critique of formalism as the only way to model understanding was already developed in 1976 by Joseph Weizenbaum. Particularly, he argues that the machine cannot make a judgment, as judgments cannot be reduced to calculation (Weizenbaum, 1976). While the illusion of cognition might be easy to achieve, something he did in his development of early conversational agents, of which the most famous is *ELIZA*, the necessary inclusion of morals and emotion of the process of judging intrinsically limit what machines can do[6]. Formal representation might provide a certain appearance of understanding, but lacks its depth.

Around the same time, however, was developed another approach to formalizing the intricacies of cognition. Warren McCullough's seminal paper, *A logical calculus of the ideas immanent in nervous activity*, co-written with Walter Pitts, offers an alternative to abstract knowledge based on the embodiment of cognition. They present a connection between the systematic, input-output procedures dear to cybernetics with the predicate logic writing style of Russell and others (McCulloch & Pitts, 1990). This attachment to input and output, to their existence in complex, inter-related ways, rather than self-contained propositions is, interestingly, rooted in his ac-

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

tivy as a literary critic[7].

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

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

A couple of decades later, Abelson and Sussman still note, in their introductory textbook to computer science, the difficulty to convey meaning mechanically:

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

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

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

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

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

3.1.2 Knowing-what and knowing-how

With the publication of Wittgenstein's *Philosophical Investigations*, there was a radical posture change from one of the logicians whose work underpinned AI research. In his second work, he disown his previous approach to language as seen in the *Tractatus Logico-philosophicus*, and favors a more contextual, use-centered frame of what language is. Rather than what knowledge is, he looks at how knowledge is acquired and used; while (formal) lanuage was previously defined as the exclusive means to translation concepts in clearly understandable terms, he broadens his perspective in the *Inquiries* by stating that language is *"the totality of language and the activities with which it is intertwined"* and that *"the meaning of a word is its use within language"* (Wittgenstein, 2004), noting context and situatedness as a important factors in the understanding process.

At first, then, it seemed possible to make machines understand through the use of formal languages. The end of the first wave of AI development, a branch of computation specifically focused on cognition, has shown some limits to this approach. Departing from formal languages, we now investigate how an embodied and situated agent can develop a certain sense of understanding.

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

Knoweldge and situation

As hinted at by the studies of McCullough and Levitt, the process of understanding does not rely exclusively on abstract logical processes, but also on the processes involved in grasping a given object, such as, in their case, what is being seen. It is not just what things are, but how they are, and how they are *perceived*, which matters. Different means of inscription and description do tend to have an impact on the ideas communicated and understood.

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

For instance, pseudo-code is a way to sketch out an algorithmic procedure, which might be considered agnostic when it comes to implementation details. At this point, the pseudo-code is halfway between a general idea and the specificity of the particular idiom in which it is inscribed. One can consider the pseudo-code in 38, which describes a procedure to recognize a free-hand drawing and transform it into a known, formalized glyph. Disregarding the implementation details means disregarding any reality

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

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

of an actual system: the operating system (e.g. UNIX or MSDOS), the input mechanism (e.g. mouse, joystick, touch or stylus), the rendering procedure (e.g. raster or vector), the programming language (e.g. JavaScript or Python), or any details about the human user drawing the circle.

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

He then follows Philip Agre's statement that a theory of cognition based on formal reason works only with objects of cognition whose attributes and relationships can be completely characterized in formal terms; and yet a formalist approach to cognition does not prove that such objects exist or, if they exist, that they can be useful. Uses of formal systems in artificial intelligence in specific, and in cognitive matters in general, is yet another instance of the map and the territory problem—programming languages only go so far in describing a problem domain without reducing such do-

main in a certain way.

Beyond the syntax of formal logic, there are different ways to transmit cognition in actionable form, depending on the form, the audience and the purpose. In particular, a symbol system does not need to be formal in order to act as a cognitive device. Logical notation exists along with music, painting, poetry and prose. In terms of form, a symbol system of formal logic is only one of many possibilities for systems of forms. In his *Languages of Art*, Nelson Goodman elaborates a theory of symbol systems, which he defines as formal languages composed of syntactic and semantic rules (Goodman, 1976), further explored in 4.1. What follows, argues Goodman, is that all these formal languages involve an act of *reference*. Through different means (exemplification, denotation, resemblance, representation), liguistic systems act as sets of symbols which can denote or exemplify or refer to in more complex and indirect ways, yet always between a sender and a receiver.

Despite the work of Shannon (Shannon, 2001) and its influence on the development of computer systems, communication, as the transfer of meaning from one individual to one or more other individuals, does not exclusively rely on the use of mathematical notation use of formal languages.

From Goodman to Goody, the format of representation also affords differences in what can be thought and imagined. Something that was always implicit in the arts—that representation is a complex and ever-fleeting topic—is shown more recently in Marchand-Zañartu and Lauxerois's work on pictural representations made by philosophers, visual artists and novelists (such as Claude Simon's sketches for the structure of his novel *La Route des Flandres*, shown in 3.1) (Marchand-Zañartu & Lauxerois, 2022). How specific domains, including visual arts and construction, engage in the relation between form and cognition is further adressed in chapter 4.

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

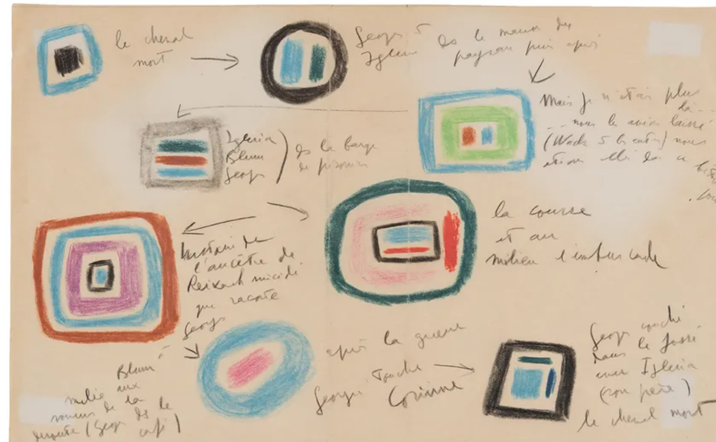

Figure 3.1: Tentative d'organisation visuelle pour le roman La Route des Flandres, années 1960 - Claude Simon, écrivain

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

Constructing knowledge

There are multiple ways to express an idea: on can use formal notation or draft a rough sketch with different colors. These all highlight different degrees of expression, but one particular way can be considered problematic in its ambition. Formal languages rely on the assumption, that all which can be known can ultimately be expressed in unambiguous terms. First shown by Wittgenstein in the two main eras various eras of his work, we know focus on the ways of knowing which cannot be explicited.

First of all, there is a separation between *knowing-how* and *knowing-that*; the latter, propositional knowledge, does not cover the former, practical knowledge (Ryle, 1951). Perhaps one of the most obvious example of this duality is in the failure of Leibniz to construct a calculating machine, as told by Matthew L. Jones in his book *Reckoning with Matter*. In it, he traces

the history of philosophers to solve the problem of constructing a calculating machine, a problem which would ultimately be solved by Charles Babbage, with the consequences that we know (Jones, 2016).

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

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

Polanyi, as a scientist himself, starts from another assumption: we know more than we can tell. In his eponymous work, he argues against a positivist approach to knowledge, in which empirical and factual deductions are sufficient to achieve satisfying epistemological work. What he proposes, derived from *gestalt* psychology, is to consider some knowledge of an object as the knowledge of an integrated set of particulars, of which we already know some features, by virtue of the object existing in an external approach. This integrated set, in turn, displays more properties than the sum of its parts. While formal notation suggests that the combination of formal symbols does not result in additional knowledge, Polanyi rather argues, against Descartes, that relations and perceptions do result in additional knowledge.

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

Grene, 1969)

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

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

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

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

This approach has been touched upon from a practical programmer's perspective in section 2.3.3, through a historical lens but it does also posses

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

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

We started our discussion of understanding by defining it as the acquisition of the knowledge of a object—be it a concept, a situation, an individual or an artfefact, which is accurate enough that it allows us to predict the behaviour of and to interact with such object.

Theories of how individuals acquire understanding (how they come to

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

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 fell short of handling everyday tasks which humans would consider to be menial. This has led us to consider a different approach to understanding, in which it is acquired through contextual and embodied means. Particularly, we have identified this tacit knowledge as relying on a social component, as well as on a somatic component.

Source code, as a formal system with a high dependence of context, intent and implementation, mobilizes both approaches to understanding. Due to programing's *ad hoc* and bottom-up nature, 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.

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

3.2 Understanding computation

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

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

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

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

3.2.1 Software ontology

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

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

Software as abstract artifact

When he coins the phrase *abstract artifact*, Nurbay Irmak addresses software partly as an abstract object, similar in his sense to Platonic entities, and partly as a concrete object which holds spatio-temporal properties (Irmak, 2012). This is based on the fact that software requires an existence as a textual implementation, in the form of source code (Suber, 1988); it is composed of files, has a beginning (start) and an end (exit); but software also represents ideas of structure and procedure which go beyond these limitations of being written to a disk, having a compilation target or an execution time. Typically, the physical aspects of software (its manifestation as source code) can be changed without changing any of the ideas expressed by the software[10].

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

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

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

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

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

This metaphor provides an accurate but limited account of the place of source code within the definition of software: the Java implementation is itself a definition implemented in a specific bytecode, while arrays in Pascal are different abstractions than arrays implemented in C, etc. Nonetheless, source code is that which gives shape to the ideas immanent in software—through a process of concretization—and which hides away the details of the hardware—through abstraction. This metaphor of *abstract artifact* thus helps to clarify the tensions within software, and to locate the specific role of source code within the different moving parts of definition, medium and model.

Software, like other artefacts, has a relation between its *functional* properties (i.e. purpose that are intended to be achieved through their use) and *structural* ones (both conceptual and physical configuration which are involved in the fullfilment of the functional purpose) (Turner, 2018). As such, it also belongs to the broader class of technology, and thus holds

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

Software as a relational object

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

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

Technology is a dynamic of organized, but inorganic matter (Stiegler, 1998). Following Latour, we also extend the conception of inorganized matter to include social influences, personal practices, and forms of tacit and explicit knowledges (Latour, 2007). That is, the ambiguity of the technical object is that it extends beyond itself as an object, entering into a relation with its surrounding environment, including the human individual(s) which shape and make use of it.

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

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

Technical objects imply another mode of being, consequential to the recognition of the limtations of magic—humanity's primary mode of being. Technicity, according to Simondon, focuses on the particular, on the elements, *a contrario* to the religious mode of being, which finds more stability in a persepective of totality, rather than a focus on individuals[11].

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

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

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

striving for unity and totality, for the balance between the objective and the subjective. Yet, rather than being a monadic unity of a single principle, Simondon considers the aesthetic mode as a unifying network of relationships [12]. He further argues that the aesthetic mode goes beyond taste and subjective preference, into a fundamental aspect of the way in which human beings relate to the world around them. An aesthetic object therefore acquires the property of being beautiful by virtue of its relationships, of its connections between the subject and the objective, between one's history and one's perceptions, and the various elements of the world, and the actions of the individual. Finally, the aesthetic thought when related to the technical object consists in preparing the communication between different communities of users, between different perspectives on the world, and different modes of action upon this world. Ultimately, the aesthetic mode of can therefore be seen as the revealing of a nexus of relationships found in its environment, highlighting the key-points of in the structure of the object [13]. How aesthetics enables a holistic thought through the use of sensual markers will be the subject of 4.

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

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

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

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

Software is a manifestation of technology as both knowledge and action. Furthermore, it also enables ways to act mechanically on knowledge and ideas, an affordance named *epistemic action* by David Kirsh and Paul Maglio (Kirsh & Maglio, 1994). They define epistemic actions as actions which facilitate thinking through a particular situation or environment, rather than having an immediate functional effect on the state of the world. As technology changes the individual's relationship to the world, software does so by being the dynamic, manipulable notion of a state of a process, ever evolving around a fixed structure, and by changing the conceptual understanding of said world (Rapaport, 2005). Such examples of world related to the environment in which software exists, e.g. the social environment, or hardware environment, or the environment which has been recreated within software. David M. Berry investigates this encapsulation of world in his *Philosophy of Software*:

> *The computational device is, in some senses, a container of a universe (as a digital space) which is itself a container for the basic primordial structures which allow further complexification and abstraction towards a notion of world presented to the user. (Berry, 2011)*

Software-as-world is the material implementation of a proposed model, itself derived from a theory. It therefore primarily acts at the level of *episteme*, sometimes even limiting itself to it[15]. Paradoxically, it is only through

highly perfected theory of computation would devolve into neither more not less than a generalized theory of the physical world. Sans *some notion of efficacy of mechanism m, conversely, no limits could be either discerned or imposed on what could be computed, evacuating the notion of constraint, and hence of intellectual substance. Freed from all strictures of efficacy or mechanism from any requirement to sustain physical realizability, computation would become fantastic (or perhaps theistic): meaning spinning frictionlessly in the void.*" (Smith, 2016).

[15]Functional programming languages take pride in the fact that they have no effect on the world around them, being composed exclusively of so-called *pure functions*, and no external side-effects, or input/output considerations.

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

Along with software's material and theoretical natures (i.e. in contemporary digital computers, it consists of electrons, copper and silicium and of logical notations), another environment remains—that of the intent of the humans programming such software. Indeed, thinking through the function of computational artefacts, Turner states that it is *agency* which determines what the function is. He defines agency as the resolution of the difference between the specification (intent-free, external to the program) and semantic interpretation (intent-rich, internal to the programmer) (Turner, 2018). In order to understand a computer program, to understand how it exists in multiple worlds, and how it represents the world, we need to give it meaning. To make sense of it, a certain amount of interpretation is required in relation to that of the computer's—such that the question "what does a Turing machine do?" has `n+1` answers. 1 syntactic, and `n` semantic (e.g. however many interpretations as there can be human interpreters) (Rapaport, 2005). In his investigation into what software is, Suber corroborates:

> *This suggests that, to understand software, we must understand intentions, purposes, goals, or will, which enlarges the problem far more than we originally anticipated. [...] We should not be surprised if human compositions that are meant to make machines do useful work should require us to posit and understand human purposiveness. After all, to distinguish* literature *from noise requires a similar undertaking. (Suber, 1988)*

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

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

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

3.2.2 Software complexity

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

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

Along with different levels of existence needed to be taken into account by the programmer, software also exhibits specific kinds of complexity. Our definition of complexity will be the one proposed by Warren

Weaver. He defines problems of (organized) complexity as those which involve dealing simultaneously with a sizable number of factors which are interrelated into an organic whole (Weaver, 1948)[16]. Specifically, there are three different types of software complexity that we look at: technical complexity, spatio-temporal complexity and modelling complexity.

Levels of software

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

The relationship between the *ideal* and the *actual* versions of the same software is not straightforward. Ideal software only provides an intent, a guidance towards a goal, assuming, but not guaranteeing, that this goal will be reached[17]

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

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

[17] A popular engineering saying is that complements this approach by stating that: *"In theory, there is no difference between theory and practice. In practice, there is."*. This quote is often mis-attributed to Richard P. Feynman or Albert Einstein, but has been traced to Benjamin Brewster, writing in the Yale Literary Magazine of 1882. (Investigator, 2018)

```
how to get the difference in character length between two words

store the first word in a variable
store the second word in a variable

store the difference between the number of characters in the first
↪  word
and the number of characters in the second word

print the difference to the console
```

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

Writing on the myths of computer science, James Moor (Moor, 1978) allows us to think through this distinction between ideal and practical along the lines of the separation between a theory and a model. The difference between a model and a theory is that both can exist independently of one another—one can have a theory for a system without being able to model it, while one can also model a system using *ad hoc* programming techniques, instead of a coherent general theory.

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

The third level at which software exists is that of hardware. While the ideal version of software is presented in natural language, diagrams or pseudo-code, and while the practical version of software exists as executable source code, software also exists at a very physical level—that of transistors and integrated circuits. To illustrate the chain of material levels at which software exist, the series of listings in 39, 40, 41 and 42 perform the exact same function of implementing a FILL ME algorithm, respectively in pseudo code, in C, in Assembler and in bytecode.

The gradient across software and hardware has been examined thoroughly (Kittler, 1997; Chun, 2008; Rapaport, 2005), but never strictly defined. Rather, the distinction between what is hardware and what is software is

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

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

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

    printf("%d", difference);

    return 0;
}
```

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

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

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

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

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

relative to where one draws the line: to a front-end web developer writing JavaScript, the browser, operating system and motherboard might all be considered hardware. For a RISC-V assembly programmer, only the specific CPU chip might be considered hardware, while the operating system being implemented in C, itself compiled through Assembly, would be considered software. A common definition of hardware, as the physical elements making up the computer system, overlooks the fact that software itself is, ultimately, physical changes in the electrical charge of the components of the computer.

Software can be characterized the dynamic evolution of logical processes, described as an ideal specification in natural languages, as a practical realization in programming languages, and in specific states of hardware components. Furthermore, the relations between each of these levels is not straightforward: the ideal and the practical can exist independently of each other, while the practical cannot exist independently of a machine. For instance, the machine on which a given program text is executed can be a *virtual machine* or, conversely, a real machine managing virtual memory.

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

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

Spatio-temporal complexity

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

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

Temporal complexity relates to the discrepancy between the way the computer was first thought of —i.e. as a Turing machine which operates linearly, on a one-dimensional tape—and further technological developments. The hardware architecture of a computer, and its specification as a Turing machine involve the ability for the head of the machine to jump at different locations. This means that the execution and reading of a program would be non-linear, jumping from one routine to another across the source code. Such an entanglement is particulary obvious in Ben Fry's Distellamap series of visualizations of source code (3.2 represents the execu-

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

Furthermore, the machine concept of time is different from the human concept, and different machines implement different concepts. For instance, operations can be synchronous or asynchronous, thus positing opposite frames of reference, since the only temporal reference is the machine itself[19]. While humans have somewhat intuitive conceptions of time as a linearly increasing dimension, computer hardware actually includes mutliple clocks, used for various track-keeping purposes and structuring various degrees of temporality (Mélès, 2017).

Later on, the introduction of multi-core architecture for central processing units in the late 2000s has enabled the broad adoption of multi-threading and threaded programming. As a result, source code has transformed from a single non-linear execution to a multiple non-linear process, in which several of these non-linear executions are happening in parallel. Keep tracking of what is executing when on which resource is involved in problems such as *race conditions*, when understanding the scheduling of events (each event every e.g. 1/18000000th of a second on a 3.0 Ghz CPU machine) becomes crucial to ensuring the correct behaviour of the software.

Conversely, the locii of the execution of software creates contributes to those issues. Even at its simplest, a program text does not necessarily exist as a single file, and is never read linearly. Different parts can be re-edited and re-arranged to facilitate the understanding of readers[20]. Mod-

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

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

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

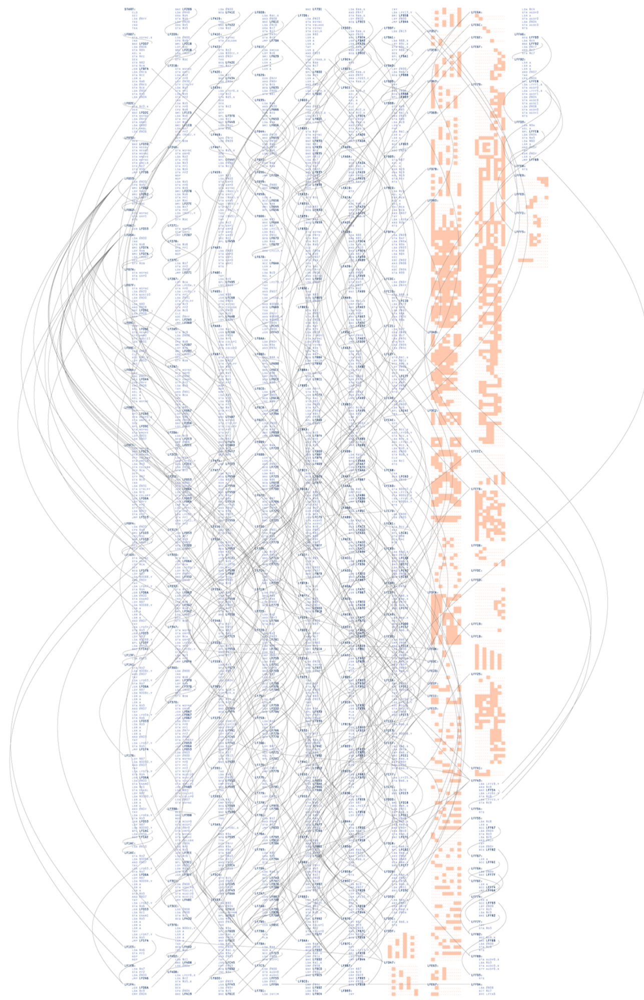

Figure 3.2: Visualization of the execution of Pac-Man's source code

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

Where exactly these files exist is not always immediately clear, as their reference by name or by Uniform Resource Locator (URL) can obfuscate whether or not a file exists on the current machine. As such, software can be (dis-)located across multiple files on a single machine, on multiple processes on a single machine, or on multiple processes on multiple machines (on a local-area or wide-area network) (Berry, 2011). Facilitating navigation between files through the references that files hold to one another is one way that the tools of the programmers alleviate cognitive burden, as we will see in 3.3.2.

Additionally, time and space in computation can interact in unexpected ways, and fragments the interface to the object of understanding. For instance, the asynchronicity of requesting and processing information from distinct processes is a spatial separation of code which has temporal implications (e.g. due to network latency). When and where a certain action takes place becomes particularly hard to follow.

Modelling complexity

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

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

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

This translation process involves the development of *models*; these are abstract descriptions of the particular entities which are considered to be meaningful in the problem domain. The process of abstracting elements of the problem domain into usable computational entities is an essential aspect of software development, as it composes the building blocks of software architectures (see 2.1.1 for discussion of software architects). Abstraction encompasses different levels, at each of which some aspect of the problem domain is either hidden or revealed, and finding the right balance of such showing or hiding in those models does not rely on explicit and well-known rules. but rather on cognitive principles. Starting from the observation that there no generalizable rules for modelling classes in computer science, Parsons and Wand suggest that cognitive principles can be a productive way foreward[21]. They base their proposal on the theories of Lakoff and Johnson, insofar as metaphors operate cognitively by mapping two entities abstracting at the same level; such a tool for understanding is further explored in 3.3.1.

For a banking system, this might involve a `Client` model, an `Account` model, a `Transfer` model and a `Report` model, among others. The ability to represent a `Client` model at a productive abstraction level is then further complicated by the conceptual relations that the model will hold with other models. Some of these relations can be made explicit or implicit, and interact in unexpected ways, since they differ from what our personal conception of what a `Client` is and of what it can do[22].

---

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

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

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

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

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

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

---

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

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

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

3.2.3 The psychology of programming

In practice, programmers manage to write, read and understand source code as a pre-requisite of producing reliable source code. Being able to write a program has for effective pre-requisite a thorough understanding of the problem, intent and platform, making the programming activity a form of applied understanding[23].

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

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

ate modification, re-use or maintenance of the software? What cognitive abilities do they summon, and what kind of technical apparatuses play a role in this process? In answering these questions, we will see how the process of understanding a program text is akin to constructed a series of mental models, populating a cognitive map.

The earliest studies of how computer programmers understand the code they are presented with consisted mostly in pointing out the methodological difficulties in doing so (Sheil, 1981; Shneiderman, 1977; Weinberg, 1998). This is mainly due to three parameters. First, programming is an intertwined combination of notation, practices, tasks and management, each of which have their own impact on the extent to which a piece of source code is correctly understood, and it is hard to clearly establish the impact of each of these. Second, program comprehension is strongly influenced by practice—the skill level of the programmer therefore also influences experimental conditions[24]. Third, these early studies have found that programmers have organized knowledge bases, if informal and immaterial. This means that, while programmers demonstrate epistemic mastery, they are limited in their ability to explain the workings of such ability.—that is, the constitution and use of their own mental models.

Marian Petre and Alan Blackwell attempted in their 1992 study to identify these mental models and their uses. They asked 10 expert programmers from North America and Europe to describe the thought process in source code-related problem-solving and design solutions in code. While this study was an investigation into the design of code, before any writing happens, one of the limitations is that it did not investigate the understanding of code, which takes places once the writing has been done (by

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

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

The main conclusion of their study is that, beyond the fact that each programmer had slightly different descriptions of their mental process, there are some commonalities to what is happening in someone's thoughts as they start to design software. The behaviour is dynamic, but controlled; the resolution of that behaviour was also dynamic, with some aspects coming in and out of focus that the will of the programmer, providing more or less uncertainty, level of details and fuzziness on-demand; and those images co-existed with other images, such that one representation could be compared with another representation of a different nature (Petre & Blackwell, 1997). Finally, while most imagery was non-verbal, all programmers talked about the need to have elements of this imagery labelled at all times, hinting at a relationship between syntax and semantics to be translated into source code.

Francoise Détienne, in her study of how computer programmers design and understand programs (Detienne, 2001), defines the activity of designing and understanding programs in activating *schemas*, mental representations that are abstract enough to encompass a wide use (web servers all share a common schema in terms of dealing with requests and responses), but nonetheless specific enough to be useful (requests and responses are qualitatively different subsets of the broader concept of inputs and outputs). An added complexity to the task of programming comes with one of the dual nature of the mental models needing to be activated: the computer's actions and responses are comprised of the prescriptive (what the computer should do) to the effective (what the computer actually does). In order to be appropriately dealt with, then, programmers must activate and refine mental models of a program which resolves this tension. To do so, they seem to resort to spatial activities, such as *chunking* and *tracing* (Cant et al., 1995), thus hinting at a need to delimitate some cognitive objects with a material metaphor, and connecting those concepts with a spatial metaphor.

In programming, within a given context—which includes goals and heuristics—, elements are being perceived, processed through existing knowledge schemas in order to extract meaning. Starting from Kintsch and Van Dijk's approach of understanding text (Kintsch & van Dijk, 1978), Détienne nonetheless highlights some differences with natural language understanding. In program texts, she finds, there is an entanglement of the plan, of the arc, of the tension, which does not happen so often in most of the traditional narrative text. A programmer can jump between lines and files in a non-linear, explorative manner, following the features of computation, rather than textuality. Program texts are also dynamic, procedural texts, which exhibit complex causal relations between states and events, which need to be kept track of in order to resolve the prescriptive/effective discrepancies. Finally, the understanding of program text is first a general one, which only subsequently applies to a particular situation (a fix or an extension needing to be written), while narrative texts tend to focus on specific instances of protagonists, scenes and descriptions, leading to broad thematic appreciation.

Conversely, a similarity in understanding program texts and narrative texts is that the sources of information for understanding either are: the text itself, the individual experience and the broader environment in which the text is located (e.g. technical, social). Building on Chomsky's concepts, the activity of understanding in programming can be seen as understanding the *deep structure* of a text through its *surface structure* (Chomsky, 1965). One of the heuristics deployed to achieve such a goal is looking out for what she calls *beacons*, as thematic organizers which structure the reading and understanding process (Wiedenbeck, 1991; Koenemann & Robertson, 1991). For instance, in traditional narrative texts, beacons might be represented by section headings, or the beginning or end of paragraphs. However, one of the questions that her study hasn't answered specifically is how the specific surface structure in programming results in the understanding of the deep structure—in other terms, what is the connection be-

tween source code syntax, programmer semantics and program behavior.

Détienne's work ushers in the concept of a mental model as means of understanding in programmers, which proved to be a fruitful, if not settled field of research. Mental models are a dynamic representation formed in working memory as a result of using knowledge from long term memory and the environment (Cañas & Antolí, 1998). As such, they are a kind of internal symbolic representation of an external reality, are a rigorous, personal and conceptual structure. They are related to knowledge, since the construction of accurate and useful mental models through the process of understanding is shaped by, and also underpins knowledge acquisition. However, mental models need not be correlated with empirical truth, due to their personal nature, but are extensive enough to be described by formal (logical or diagrammatical) means. Mental models can be informed, constructed or further qualified by the use of metaphors, but they are nonetheless more precise than other cognitive structures such as metaphors—a mental model can be seen as a more specific instance of a conceptual structure than a metaphor.

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

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

the result of the actions taken with the current state of the idea(s) held.In their work on computer-enabled cognitive skills, Kirsh and Maglio develop on the use of epistemic actions:

> *More precisely, we use the term epistemic action to designate a physical action whose primary function is to improve cognition by:*
>
> 1. *reducing the memory involved in mental computation, that is, space complexity;*
> 2. *reducing the number of steps involved in mental computation, that is, time complexity;*
> 3. *reducing the probability of error of mental computation, that is, unreliability.*
>
> *(Kirsh & Maglio, 1994)*

Since epistemic actions rely on engaging with a text, at the syntax and semantics level, it has often been assumed by programmers and researchers that reading and writing code is akin to reading and writing natural language. Additional recent research in the cognitive responses to programming tasks, conducted by Ivanova et. al., do not appear to settle the question of whether programming is rather dependent on language processing brain functions, or on functions related to mathematics (which do not rely on the language part of the brain) (Ivanova et al., 2020), but contributes empirical evidence to that debate. They conclude that, while language processing might not be one of the essential ways that we process code—excluding the *code is text* hypothesis—, it also does not rely on exclusively mathematical functions. Stimulating in particular the so-called multi-demand system, it seems that programming is a polymorphous activity involving multiple exchanges between different brain functions. What this implies, though, is that neither literature, linguistics nor mathematics should be the only lens through which we look at code.

In a way, then, programming is a sort of fiction, in that the pinpointing of its source of existence is difficult, and in that it affords the experience of imagining contents of which one is not the source, and of which the certainty of isn't defined, through a particular syntactic configuration. Both programming and fiction suggest surface-level guiding points helping the process of constructing mental models as a sort of conceptual representation. It is also something else than fiction, in that it deals with concrete issues and rational problems [25], and that it provides a pragmatic frame for processing representations, in which assumptions stemming from burgeoning mental models can be easily verified or falsified, through the taking of epistemic actions. It might then be appropriate to treat it as such, simultaneously fiction and non-fiction, as knowledge and action, mathematic and artistic. Indeed, it is also an artistic activity which, in Goodman's terms, might be seen as *an analysis of [artistic] behavior as a sequence of problem-solving and planning activities.*" (Goodman & Others, 1972).

Remains the interpretation issue mentioned above: the interpretation of the machine is different from the interpretation of the human, of which there are many, and therefore what also needs to be intepreted is the intent of the author(s). Such a tension between the computer's position as an extremely fast executer and the programmer's position as a cognitive agent is summer up by Niklaus Wirth in *Beauty Is Our Business*, Dijkstra's *festschrift*: "*What the computer interprets, I wanted to understand.*" (Wirth, 1990).

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

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

[25]More often than not, a pestering bug

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

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

3.3 Means of understanding

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

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

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

3.3.1 Metaphors in computation

Our understanding of metaphors relies on the work of George Lakoff and Mark Johnson[26] due to their requalification of the nature and role of metaphor beyond an exclusively literary role. While Lakoff and Johnson's approach to the conceptual metaphor will serve a basis to explore these linguistic devices as a cognitive means across software and narrative, we also argue that Ricoeur's focus on the tension of the *statement* rather than primarily on the *word* will help us better understand some of the aesthetic manifestations and workings of software metaphors. Following a brief overview of their contributions, we then examine the various uses of metaphor in software, from end-users to programmers.

Theoretical background

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

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

Lakoff and Johsnon's seminal work develops a theory of conceptual metaphors by highlighting their essential dependence on pre-existing cognitive structures, which we associate with already-understood concepts. The metaphor maps a source domain (made up of cognitive structure(s)) to a target domain. In the process, they extend the field of applicability of metaphors from the strictly literary to the broadly cultural: metaphors work because each of us has some conception of those domains involved in the metaphorical process.

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

Starting with the role of metaphors manifested in expressions such as *the desktop*, *the mouse*, or *the cloud* for end-users, we will then turn to the programmers relationships to their environment as understood metaphorically. The relationship between poetic metaphor and source code will be developed in 5.2.2; with the topic of syntax and semantics in programming languages in 5.1.1, we will see that metaphor-induced tensions can be a fertile ground for poetic creation through aesthetic manifestations.

Metaphors for end-users

It is interesting to consider that the first metaphor in computing might be concommitant with the first instance of modern computing—the Turing *machine*. While Turing machines are widely understood as being manifested into what we call digital computers (laptops, tablets, smartphones, etc.), and thus definitely within the realm of mechanical devices, the Turing machine is not strictly a machine *per se*. Rather, it is more accurately defined as a mathematical model which defines an abstract machine. Indeed, as we saw in 3.2.1, computers cannot be proven or assumed to be machines, because their terminology comes from logic, textual, or discursive traditions (e.g. reference, statement, names, recursion, etc.) and yet they are still *built* (Smith, 1998). Humans can be considered Turing machines (and, in fact, one of the implicit requirements of the Turing machine is that, given enough time and resources, a human should be able to compute anything that the Turing machine can compute), and non-humans can also be considered Turing machines[27]. Debates in computer science related to the nature of computing (Rapaport, 2005) have shown that computation is far from being easily reduced to a simple mechnical concern, and the complexity of the concept is perhaps why we ultimately revert to metaphors in order to better grasp them.

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

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

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

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

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

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

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

Galloway productively deploys this heuristic in the context of interfaced computer work: cognitive mapping is the process by which the individual subject situates himself within a vaster, unrepresentable totality, a process that corresponds to the workings of ideology[28]. Here, we can see how metaphors can act as both cognitive tools to make sense of objects, but also as obfuscating devices to cloak the reality of the environment[29]. The cognitive processes enable by metaphors help provide a certain sense of the unthinkable, of that which is too complex to grasp and therefore must be put into symbols (words, icons, sounds, etc.).

Nielsen and Gentner develop on some challenges that arise when one uses metaphors not just for conceptual understanding, but for further conceptual manipulation. In *The Anti-Mac Interface*, they point out that differences in features between target domain and source domain are inevitable. For instance, a physical pen would be able to mark up any part of a physical form, whereas a tool symbolize by a pen icon on a document editing software might restrict an average user to specific fields on the form. Their study leads to assess alternatives to one kind of interface[30], in or-

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

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

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

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

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

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

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

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

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

Programmer-facing metaphors

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

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

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

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

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

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

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

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

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

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

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

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

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

3.3.2 Tools as a cognitive extension

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

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

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

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

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

One of the first interfaces for writing computer code included the text editor called *EMACS* (an acronym for *Editor MACroS*), with a first version released in1976. Containing tens of thousands of commands to be input by the programmer at the surface-level in order to affect the deeper level of the computing system, *EMACS* allows for remote access of files, modeful and non-linear editing, as well as buffer-based manipulation *Vim* (Greenberg, 1996). This kind of text editor acts as an interfacing system which allows for the almost real-time manipulation of digitized textual objects.

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

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

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

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

IDEs therefore solve some of the mental operations performed by programmers when they engage with source code, such as representing code blocks through proper indentations. The automation of tooling and workflow increased in software such as Eclipse, IntelliJ, NetBeans, WebStorm Visual Studio Code[31] has led to further entanglements of technology and appearance. By organizing code space through actions such as self documentation, folding code blocks, finding function declarations, batch reformatting and debug execution, they facilitate cognitive operations such as chunking, tracing, or highlighting beacons (Bragdon et al., 2010). These technical features show how a tool which operate at primarily the aesthetic level has consequences on the understandability of the system represented, even though this is, again, dependent on the skill level of the programmer (Kulkarni & Varma, 2017).

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

[31]Through which this thesis is written.

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

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

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

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

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

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

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

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

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

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

Extended cognition

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

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

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

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

The computer therefore supports epistemic actions through its use of metaphors (to establish a fundamental base of knowledge) and of actions (to probe and refine the validity of those metaphors) to build a mental model of the problem domain. In the case of IDEs, the problem domain is the source code, and these interfaces, by allowing means of scanning and navigating the source code, are part of what Simon Penny calls, after Clark and Chalmers, *extended cognition* (Penny, 2019). Extended cognition posits that our thinking happens not only in our brains, but is also located in the tools we use to investigate reality and to deduce a conceptual model of this reality based on empirical results. We consider IDEs a specific manifestation of embodied cognition, actively helping the programmer to define, reason about, and explore a code base. The means of taking epistemic action, then, are also factors in contributing to our understanding of the program text at hand. In this spirit, David Rokeby goes as far as qualifying the computer as a *prosthetic organ for philosophy*, insofar as it helps him formulate accurate mental models as he interacts with them through computer interfaces, compensating for its formal limitations[32].

This brings us back to our discussion of Simondon's technical and aesthetic modes of existence 3.2.1. As highlighted by the use of software tools in the sense-making process of a program text, formal syntax only operates on distinct, fragmented concepts, as evoked in the technological mode[33].

---

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

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

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

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

We pointed out the role that metaphors play in creating connections between pre-existing knowledge and current knowledge, building connections between both in order to facilitate the construction of mental models of the target domain. Metaphors are used by programmers at a different level, helping them grasp concepts (e.g. memory, objects, package) without having to bother with details. As we will see in the following chapters (see 4.2.1 and 5.2.2), metaphors are also used by programmers in the source code they write in order to elicit this ease of comprehension for their readers.

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

---

coding." (Rokeby, 2003)

> *Code is therefore technical and social, and material and symbolic simultaneously. Rather, code needs to be approached in its multiplicity, that is, as a literature, a mechanism, a spatial form (organization), and as a repository of social norms, values, patterns and processes. (Berry, 2011)*

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

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

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

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

Chapter 4

Beauty and understanding

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

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

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

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

4.1 Aesthetics and cognition

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

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

4.1.1 Source code as a language of art

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

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

Goodman develops a systematic approach to symbols in art, freed from any media-specificity (e.g. f from clocks to counters, from diagrams to maps models, from musical scores to painters' sketches and linguistic scripts). A symbolic system, in his definition, consists of characters, along with rules to govern their combination with other characters, itself correlated with a field of reference. These symbols and their arrangement within a work of art supports an aesthetic experience[1] and, since they are syntactic system which operate at the semantic level, they can be rigorous communicative systems.

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

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

perience[2]. 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.

Third, the requirement of syntactic repleteness demands that relatively fewer factors need to be taken into account during the interpretative process[3]. On one hand, we can consider that any additional aspects of the source code (such as the display font or the syntax highlighting discussed in 3.3.2) are ultimate irrelevant to the computer, thus making it a poorly replete symbol sytem. On the other hand, the importance of such factors, along with abilities to write a program with the same function but with different syntax, pleads for a relatively replete syntactical system. The tendency of program text to veer towards verbosity indeed implies this desir-

[2]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)

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

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

Finally, semantic density refers to whether or not there is a limit to the amount of concepts that the symbol system can refer to. As we have shown in 45, the affordances that programming languages provide to represent phenomena and concepts from the problem domain fulfill this requirement. While we have been previously concerned with syntax, this ability of programming languages to refer to a problem domain which has not yet shown its limitations at the semantic level is one which gives it representational power beyond strict computational concepts.

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

It thus seems like source code satisfies to a large extent the critieria to be a language of art, meaning that it exhibits some of the properties which tend to elicit an aesthetic feeling. Most notably, it does not possess a very dense syntax, nor can it be considered replete both from the perspective of the computer and of the human[4], but it nonetheless refers possesses a

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

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

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

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

The labelling process therefore serves as the symbolic expression for a particular field. In source code, this can happen through variable naming, but also through type definition[5], as well as additional affordances which we look at in 5.2, such as the layering of semantic references and the establishment of habitable cognitive structures.

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

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

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

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

class Interest {
    int priority;
    String name;
}
```

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

used—the how of the program, rather than the why. Indeed, a program text is a description of how to solve a problem from the computer's perspective, written extensively in machine language[6]. All source code can therefore be said to be a description of a combination of states (data) and actions (functionality).

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

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

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

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

Exemplification is another aspect of Goodman's theory, which has nonetheless remained somewhat limited (Elgin, 2011). A symbol exemplifying, also called an examplar, is considered as a stand-in for an item in the field of reference. We have seen source code act as an example in 2.1.3, where a particular program text is written in order to stand in for a broader concept. For instance, a program text can, at a lower level, exemplify a particular kind of procedure, such as encapsulation (see 45) or nestedness. The program text therefore exemplifies the constitutive element of the linked list[7]. However, a similar program text can also be an example of cleanliness, of clarity, or elegance. A program text written by a software developer can be seen as possessing the property of cleanliness (see 28 in 2.2.2), by virtue of its implementation of syntactic and semantic rules, while another program text written by a hacker can be seen as highlighting detailed hardware knowledge s(ee 30 in 2.2.2).

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

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

which give the exemplification of program texts its aesthetic potential.

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

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

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

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

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

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

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

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

event as a reliable symptom of the aesthetic.

Max Black initiates a view of metaphors which go beyond a simple comparison; dubbed the *interaction view*, he considers the metaphorical device as containing positive cognitive content, rather than simply entertaining or limiting (Black, 1955). Against a traditional view of metaphor being a word which stands in for another, Black reveals a large web of interactions which prove harder to disentangle, beyond usual similarities between two words[8]. Simply paraphrasing a metaphor, even if one captures precisely the same connotations/associations as the metaphor, does not convey the same meaning as the metaphor itself. For instance, saying *"Je chavire dans l'embrun des phénomènes"*[9] (Beckett, 1982) does not have the similar expressive power as listing all the properties of *phénomènes*. The use of the verb capsize in conjunction with spray relates to the domain of navigation, while capsize alone tends more to a dynamic movement, and spray to uncertainty and bluriness of shape. Phenomenas of the world are all requalified in the light of these new kinetic and perceptual associations.

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

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

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

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

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

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

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

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

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

4.1.2 Contemporary approaches to art and cognition

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

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

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

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

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

It is in the content approach that Carroll finds the most satisfying answer to what the aesthetic experience is. Content, here, is defined as the significant forms being apprehended, along with its combinations, juxtapositions and comparisons with other forms[10]. When we engage with the sensual aspects or an object, our attention is indeed directed first and foremost at what the object looks like, rather than how it makes one feel, or what value system it belongs to. More specifically, Carroll notes, if attention is directed with understanding to the form of the art work or to its expressive and aesthetic properties or to the interaction between those features, then the experience is said to be aesthetic (Carroll, 2002).

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

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

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

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

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

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

observation in scientific studies which segment an aesthetic experience in sequential steps (Leder et al., 2004). These evolve form perception, to implicit classification, explicit classification, cognitive mastering and fianlly evaluation—that is, fully-qualified aesthetic judgment. This conception is concomittant to Rebert et. al.'s proposal for an aesthetic framework based on processing fluency, which they define as a function of the perceiver's processing dynamics: the more fluently the perceiver can process an object, the more positive is her aesthetic response (Reber et al., 2004). While they focus their study on perceptual fluency, tending to traditional aesthetic features such as symmetry, contrast and balance; they also consider conceptual fluency as an influence on the aesthetic experience, through the attention given to the meaning of a stimulus and the relation of form to semantic knowledge structures. Such a conceptualizing thus hints at a similar skill-based, contextual framework which we have seen emerge in the aesthetic judgment of source code, and yet an additional establishment of a relation between truth and beauty[11].

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

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

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

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

> *making 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)*

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

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

4.2 Literature and understanding

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

4.2.1 Literary metaphors

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

As we develop in 5.2.2 in our reading of 74, not all of the evocative meaning of the poem are contained exclusively in each token, and the power of the whole is greater than the sum of its parts.

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

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

```
title = 'Constitution of the United States'

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

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

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

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

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

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

Listing 47: Cynical American Preamble, by Michael Carlisle, published in code::art #0 (Brand, 2019)

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

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

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

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

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

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

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

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

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

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

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

4.2.2 Literature and cognitive structures

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

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

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

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

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

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

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

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

Depiciting movement, vision, tactility and other embodied sensations allows us to crystallize and verify the work of the imaginative process. As such, literature unleashes our imaginary by recreating sensual experiences—Lakoff even goes as far as saying that we can only imagine abstract concepts if we can represent them in space[15]. It seems that

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

```
class love{};

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

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

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

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

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

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

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

Vilem Flusser considers poetic thinking as a means to bring concepts into the thinkable, and to crystallize thoughts which are not immediately available to us[16]; through various linguistic techniques, poetry allows us to formulate new concepts and ideas, and to shift perspectives. Rendered meaningful via this code poem, a certain conception of love is therefore depicted here as an exception that must be handled (with care) , and the use of a particularly dynamic keyword elicits such a feeling in a reader who previously had to throw and handle exceptions.

Another example of how source code can communicate concepts can be seen in 49. In this case, we can see in the relation between the name of the function, `find` and the three local variables `high`, `low` and `probe`, that the act of finding is going to imply some sort of *search space*. The search space is going to be traversed in an alternating way, called the *binary search* in computer science terms[17].

---

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

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

```java
package binary;

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

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

Here, we thus have two indicators, syntactical and structural. First, `high` and `low`, imply the space in-between, a space to be explored via `probe`[18]. Second, the use of only two statements inside the `while` loop represents the simplicity of the search process itself, a search process which, as `(high - low > 1)` tells us, implies a shrinking search space[19].

published versions contain bugs." (Bray, 2007)

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

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

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

4.2.3 Words in space

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

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

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

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

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

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

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

To focus on the specific tokens denoting space, we rely on the distinction operated by Marie-Laure Ryan on the topic (Ryan, 2009). The starting point she offers is to consider how the spatial extension of the text, its existence in a certain number of dimensions[20] impacts the readers' perception of the narrative.

At the simplest level, we see this illustrated in 50. In this listing, we can see how the most direct spatial perceptions of the program text, its indentation, actually represents semantic properties: the indent on `class_space` is related to it existing at a different level (scope) than the variables `dimensions` and `alone`, just like the indent before `def __init__` differs from the one before `def new_space` also signify changes in lexical scope.

Moving beyond this immediately visual spatial component, Ryan shifts to the spatial form of the text. Rather than looking at the space in which it is deployed, it is considering

a type of narrative organization characteristic of modernism that

[20] An oral narrative exists in zero dimensions, a live TV news ticker exists in one dimension, a printed or digital page exists in two dimensions, while a theater play exists in three dimensions.

```
class Space:
    class_space = "ever present"

    def __init__(self, dimensions):
        self.dimensions = dimensions
        alone = True

def new_space():
    new = new Space(4)
```

Listing 50: This snippet shows how the spatial extension of the text corresponds to the structural semantics of the code, in the Python programming language.

> *deemphasizes temporality and causality through compositional devices such as fragmentation, montage of disparate elements, and juxtaposition of parallel plot lines. (Ryan, 2009).*

Narrative, in its traditional sense of coherent, sequential events whose developments involve plot and characters, is seldom mentioned in writing source code. In source code, narrative is already deemphasized and the spatial form of the text mentioned above is therefore better suited to match the material of the code. Indeed, Ryan continues:

> *The notion of spatial form can be extended to any kind of design formed by networks of semantic, phonetic or more broadly thematic relations between non-adjacent textual units. When the notion of space refers to a formal pattern, it is taken in a metaphorical sense, since it is not a system of dimensions that determines physical position, but a network of analogical or oppositional relations perceived by the mind. (Ryan, 2009)*

Space, along with interactivity, is a core feature of the digital medium[21]. Janet Murray also puts spatiality as one of the core distinguishing features

[21] As N. Katherine Hayles states in her eponymous essay, *"print is flat, code is deep"* (Hayles, 2004)

of digital media, at the forefront of which are digital games[22].

An example of this intertwining of flat textual screen and spatial depth is the overall genre of interactive fiction, which displays prompts for textual interaction on a screen, accompanied with the description of where the reader is currently standing in the fictional world. Exploration can only be done in a linear fashion, entering one space at a time; and yet the system reveals itself to contain spaces in multiple dimensions, connected by complex pathways and relationships. The listing in 51 shows how the execution processes of a program text can be expressed spatially in the comments, and then textually in the rest of the file. Since comments are ignored by the computer, this depiction is only to help the human reader in their spatial representation of the executed program.

As Murray mentions, these features are not limited to those playful interactive systems presented as works to be explored (be it e-literature or digital games), but are rather a core component of digitality. Beyond the realm of fiction, one can see instances of this in the syntax used in both programming languages and programming environments (see 3.3.2 and our overview of IDEs). For instance, the use of the `GOTO` statement in BASIC, of the `JMP` and `MOV` instructions in x86 Assembly, or the use of the `return` in the C family of programming languages all hint at movement, at going places and coming back, representing the non-linear perception of program execution[23].

And yet, Ryan hints at anothe aspect of spatial form specifically in the

[22] *"The computer's spatial quality is created by the interactive process of navigation. We know that we are in a particular location because when we enter a keyboard or mouse command the (text or graphic) screen display changes appropriately.* (Murray, 1998)

[23] In the meantime, program execution is still considered to be linear by the machine, since instructions are executed one after the other. The use of multi-core architecture and parallel processing does complicate this picture, but programmers rarely engage directly with the specification of which CPU core executes which instruction. What they do engage with, is parallel programming, in which things happen simultaneously, thus presenting cognitive complexity insofar as two processes being run in parallel imply some sort of distinct semantic spaces to be reflected in the mental model of the programmer.

```
* Part 1 -- Initial checks
*
*      * . called by
*      |   MAC clients
*      v                      . . No
* +---------+  +------------+ .   +--------------------+
↪ +====================+
* | mac_tx |->| device     |-*-->| mac_protect_check |->v Is this
↪ the simple v
* +---------+  | quiesced? |     +--------------------+   v case? See
↪ [1]      v
*              +------------+           |
↪ +====================+
*                   * . Yes             * failed                   |
*                   v                   | frames                   |
*             +---------------+         |
↪ +--------+----------+
*             | freemsgchain |<----------+          Yes . *
↪  No . *
*             +---------------+                          v
↪       v
*                                                  +------------+
↪ +---------+
*                                                  |   goto     |
↪ |  goto   |
*                                                  |   Part 2   |
↪ | SRS TX |
*                                                  | Entry [A] |
↪ |  func   |
*                                                  +------------+
↪ +---------+
*                                                        |
↪       |
*                                                        |
↪       v
*                                                        |
↪ +---------+
*
↪ +----------->| return |
*
↪ | cookie |
*
↪ +---------+
```

Listing 51: This listing includes an execution flow diagram inside the program text itself, testifying to the inherently fragmented and non-linear execution of source code. (Mustacchi, 2019)

digital medium:

> *But an even more medium-specific type of spatial form resides in the architecture of the underlying code that controls the navigation of the user through a digital text. (Ryan, 2021)*

As writers and readers of this architecture, of which source code is the blueprint, we gather information through syntax about developments in space and time into a cognitive map or mental model of narrative space[24].

Mental maps are therefore dynamically constructed in the course of reading and consulted by the reader to orient herself in the program. A very simple example of spatialization of meaning, both visually and conceptually, can be seen in 52. There, the spatial component is rendered specifically through the syntax of HTML. HTML, as a markup language, has a specific ontology: it is fundamentally made up of elements who contain other elements, or are self-contained. When an element is contained into another, a specific semantic relationship occurs, where the container influences the contained, and vice-versa. Therefore, what we see at first is layout spatialization, which leads to this specific triangle shape. By using the semantics of the language, in which certain elements can only exist in the context of others, this layout spatialization[25] also comes to delimit certain semantic areas. This explicitly poetic example takes religion, and the representation of God as its problem domain; its expressive force comes by describing it as both the all-including and the all-included, and thus escaping the implicit rules of everyday spatiality, that a thing cannot contain itself.

A more concrete example can be seen in 53. Written in the style of soft-

[24]The term *narrative* is used here to describe the effective behaviour of the program, once executed. Since source code appreciation is subject to its function, following the narrative of source code would then amount to following its correct execution path(s), even though *description* fits better to most program texts since, from the machine perspective, it describes exactly what it is doing.

[25]While not functionally necessary, the indents added to the listing further highlight the computational concept of nestedness through visual cues.

```
<GOD>
    <universe>
        <galaxy>
            <solarsystem>
                <earth>
                    <island>
                        <town>
                            <garden>
                                <flowerbed>
                                    <snowdrop>
                                        <petal>
                                            <molecule>
                                                <proton>
                                                    <quark>
                                                        <GOD>
                                                    </quark>
                                                </proton>
                                            </molecule>
                                        </petal>
                                    </snowdrop>
                                </flowerbed>
                            </garden>
                        </town>
                    </island>
                </earth>
            </solarsystem>
        </galaxy>
    </universe>
</GOD>
```

Listing 52: Nested, by Dan Brown and published in {code poems} (Bertram, 2012)

```
func (h *http2Listener) Shutdown(ctx context.Context) error {
	pollIntervalBase := time.Millisecond
	nextPollInterval := func() time.Duration {
		// Add 10% jitter.
		//nolint:gosec
		interval := pollIntervalBase + time.Duration(weakrand
		↪ .Intn(int(pollIntervalBase/10)))
		// Double and clamp for next time.
		pollIntervalBase *= 2
		if pollIntervalBase > shutdownPollIntervalMax {
			pollIntervalBase = shutdownPollIntervalMax
		}
		return interval
	}

	timer := time.NewTimer(nextPollInterval())
	defer timer.Stop()
	for {
		if atomic.LoadUint64(&h.cnt) == 0 {
			return nil
		}
		select {
		case <-ctx.Done():
			return ctx.Err()
		case <-timer.C:
			timer.Reset(nextPollInterval())
		}
	}
}
```

Listing 53: This listing represents the various steps taken in order to shutdown a HTTP server, and shows multiple aspects of spatio-temporal complexities (WeidiDeng, 2023)

ware engineers, rather than poets, this listing describes a function which gracefully shuts down a HTTP server. Essentitlly, the function `Shutdown()` regularly checks if the number of connections to the server is zero. If it reaches zero, it considers the process completed without errors; it waits until it receives an error from the context, or if it receives a tick from a timer setup in advance.

The first reference we can look at is mostly spatial, and takes place at the declaration of `nextPollInterval`. By being another function declaration, it is both self-contained, but also has access to variables in its declarative environment, such as `pollIntervalBase`. A long, dynamic series of state-

ments which double a timer interval everytime it is called is thus compressed into a single token, `nextPollInterval`, and can then be passed as an argument to timer functions. Here, the space of the timer interval's calculation is compressed and abstracted away.

Interestingly, we can note the comment `// Add 10% jitter`, which explains the calculation of the subsequent interval. The word jitter usually refers to a quicky, jumpy movement, but is here used to facilitate the understanding of adding a random number to the previous one, effectively deviating the timer from its linear increase. Here, using the word *jitter* immediately evokes feeling of small, unpredictable change.

The second reference is primarily temporal. The keyword `defer` in the line `defer timer.Stop()` specifically marks the deferred execution of this particular function to the specific moment at which the current function (`Shutdown()`) returns. This reference is not absolute (as is the timer on the line above, even though it might not be determinate), but rather relative, itself dependent on when the current function will return. Here, the programming language itself makes it simple to express this relative temporal operation.

Finally, we can take a look at both the last `select` statement of the function to see a more complex interplay of both space and time. There are two things happening there. With the specific `<-` arrow, the pictorial representation shows how a message is incoming, either from `ctx.Done()`, which itself comes from outside the current function, given as an argument, or from `timer.C`, which comes from the timer that has just been declared in the current function. Both of these messages come from different places, one very distant, and the other very local, and might arrive at different moments. Here, the `<-` denotes the movement of an incoming message, expliciting where the messages come from, and in which order they should be treated, and thus facilitates the handling of event with varying spatio-temporal properties.

The listing 53 shows not only different spaces of executions, nor only

different moments of execution, but very much the intertwining of space and time. One of the earlier approaches to the specific tokens which represent space in the traditional novel has also related it to time: the chronotope is described by Mikhail Bakhtin as the tight entanglement of temporal and spatial relationships that are artistically expressed in literature. Those markers execute a double function, as they allow for the reification of temporal events and spatial settings during the unfolding of narrative events[26].

While Bakhtin introduces the concept from a marxist-historical point of view, analyzing notions of history, ideal, epics and folklore through that lense, it is nonetheless useful for our purposes. Chronotopes are a kind of marker which enable the understanding of where something comes from (such as an explicit module declarations in header files, or inline before a function call), or when something should happen (such as the `async/await` keyword pair in ECMAscript denoting the synchronicity of an operation or the `defer` keyword indicating that a specificied function will only be called *when* the current function returns).

Thus, the chronotopes give flesh to the events described in (and then executed from) source code. As such, they function as the primary means of materializing time in space. From a network of these chronotopes, along with metaphors and other devices that are explicited in 5.2, emerges a concretization of representation which the reader can use to constitue a mental model of the program text.

Syntactical literary devices allow readers to engage cognitively with a particular content; they enable the construction of mental models a particular narrative, through a network of metaphors, allusions, ambiguous

[26]"*Time, as it were, thickens, takes on flesh, becomes artistically visible; likewise, space becomes charged and responsive to the movements of time, plot and history.*" (Bakhtin, 1981)

interpretations and markers of space and time. We have shown that these literary devices also apply to source code, especially how the use of machine tokens and human interpretation suggest an aesthetic experience through metaphors, and with particular markers that are needed to make sense of the time and space of a computer program, which differs radically from that of a printed text. This making sense of a foreign time and space is indeed essential in creating a mental map of the storyworld (in fiction) or the world of reference (in non-fiction).

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

4.3 Architecture and understanding

At its most common denominator, architecture is concerned with the gross structure of a system. At its best, architecture can support the understanding of a system by addressing the same problem as cognitive mapping does: simplifying our ability to grasp large system. This phrase appears in Kevin Lynch's work on *The Image of the City*, in which he highlighted that our understanding of an urban environment relies on combinations of patterns (node, edge, area, limit, landmark) to which personal, imagined identities are ascribed. The process is once again that of abstraction, but goes beyond that, and includes a subjective perspective (Lynch, 1959). Moving from the urban planner's perspective to the architects, we see how each individual component contributes to the overall legibility of the system. This section

considers how individual structures, through their assessed beauty, offer a cognitive involvement to their participants.

Beauty in architecture is one of the discipline's fundamental components, dating back to Vitruvius's maxim that a building should exhibit *firmitas, utilitas, venustas*—solidity, usefulness, beauty. And yet in practice, beauty, or the abillity to elicit an aesthetic experience, is not sufficient, and sometimes not even required, for a building to be considered architectural. Even though architecture is usually considered as an art, it is also a product of engineering, and thus a hybrid field, one where function and publicness modulate what could be otherwise a "pure" aesthetic judgment.

This sections looks at architecture through its multiple aspects, to highlight to which extent some of these are reflected in source code[27]. Through an investigation of the tensions and overlaps of form, function, context and materiality in the built space, we identify similarities in the programmed space. Particularly, we will look at how an understanding of patterns translates across both domains, in response to both architecture and programming's material constraints, due to the physical instantiation of buildings and programs in a situated context.

4.3.1 Form and Function

Particularly, our interest here is with the cognitive involvement in the architectural work. What is there to be understood in a building, and how do buildings make it intelligible? The early theoretical answers to this question is to be found in the work of Italian architects, such as Andrea Palladio, whose conception of its discipline came from ideal platonic form, and mathematical relation between facade and inner elements, as well as Leon Battista Alberti, whose consideration of beauty in architecture, as such an organization of parts that nothing can be changed without detriment to

[27] Recall how, in 2.3.3, programmers tended to refer extensively to themselves as architects, engineers or craftspeople.

the whole (Scruton, 2013)[28].

While structure is meant to stand the test of time and natural forces[29], utility can be assessed by the extent to which a building fulfills its intended function. How the beauty of a building relates to its function, whether it can be completely dissociated from it, or if it is dependent on the fulfillment of its function, is still a matter of debate between formalists and functionalists. Nonetheless, the position we take here is in line with Parsons and Carlson, in that fitness of an object is a core component of how it is appreciated aesthetically (Parsons et al., 2012), and that form is hardly separable from function.

In some way, then, form should be able to communicate the function of a building. Roger Scruton, in his philosophical investigation of architecture, brings up the question of language—if buildings are to be cognitively engaged with, then one should be able to grasp what they communicate, what they stand for, what they express. To do so, he starts from the fact that architectural works are often composed of interconnected, coherent sub-parts, which then contribute to the whole, in a form of *gestaltung*.

> *Architecture seems, in fact, to display a kind of 'syntax': the parts of a building seem to be fitted together in such a way that the meaningfulness of the whole will reflect and depend upon the manner of combination of its parts. (Scruton, 2013)*

Yet, he develops an argumentation which suggests that architecture is not so much articulated as a language, than as a set of conventions and rules, and that it is not a representative medium (which would imply valid and invalid syntax, as well as intent), but rather an expressive one. Architectural significance, then, relies on the presence and arrangement of

[28] Such a definition is a reminiscent of how Vladimir Nabokov defines beauty in literature: "*A really good sentence in prose should be like a good line in poetry, something you cannot change, and just as rhythmic and sonorous*" (Nabokov, 1980)

[29] A purpose exemplified by the still standing structures of Roman and Greek antiquities, resulting from a particular mixing process of concrete.

those evolving conventions—that is, a style—rather than on the depiction of a subject through an exact syntax. While architecture might not represent content the same way literature does, it is nonetheless expressive, and relies on particular styles—recurring formal patterns and ways of doing—to express a tone, a feeling, or a *stimmung* in their dwellers.

As identified in 2.3.3, the similarities between software and architecture can be mapped as symmetrical approaches: as top-down or bottom-up, from an architect's perspective, or from a craftperson's. Since we focus on what a building expresses, we need to consider the source of such an expression. First, we look at how modernism, and the conventions that make up this architectural thought, are the top-down result of the intersection of function, form and industry, and reveal the influence of functional design on the aesthetic appreciation of a work.

The central modern architectural standard is Louis Sullivan's maxim that *form follows function*, devised as he was constructing the early office buildings in North America. Sullivan's statement is thus that what the building enables its inhabitants to do, inevitably translates into concrete, visible, and sensual consequences.

> *All things in nature have a shape, that is to say, a form, an outward semblance, that tells us what they are, that distinguishes them from ourselves and from each other ...It is the pervading law of all things organic and inorganic, of all things physical and metaphysical, of all things human and all things superhuman, of all true manifestations of the head, of the heart, of the soul, that the life is recognizable in its expression, that form ever follows function. This is the law. (Sullivan, 1896)*

The value of the building is therefore derived from what it allows the individuals to do: the office building allows them to work, the school to learn, the church to pray and the house to live. To do so, modernist architecture rejects any superfluous decoration, or extraneous addition, as

a corruption of the purity of the building's function. In a similar vein, Le Corbusier, another fundamental actor of modern architecture, equates the building with its function, advocating for the suppression of decorative clutter and unnecessary furnishings and possessions, and hailing transparency and simplicity as architectural virtues (Le Corbusier & Saugnier, 1923), and culminating in Le Corbusier's assessment that the architectural plan as a generator, and the house as a machine to be lived in.

From this perspective, architectural works are a kind of system, in that they constitute sets of interrelated structural components, where the parts are connected by distinctive structural and behavioral relations; and yet the set of conventions to which Le Corbusier contributes is an abstract representation of this systemic nature. He focuses on the plan as the primary source of architectural quality. For software developers, the equivalent of an architectural plan would be a modelling system such as UML: a language to describe structural relationships between software components, with an example shown in 4.1. From a modernist angle, the aesthetic value of a building is thus directly dependent on how well it performs an abstractly defined function for its users, assessed at a structural level.

Just as a two-dimensional floorplan and a three-dimensional building are different, a diagram and a program text are also different. This difference is highlighted throught the process of construction in architecture, and implementation in software development, involving respectively engineers and programmers to realize the work that has been designed by the architect.

It is clear the modernists thought of function as engineering function, and aligned it with engineering aesthetics[30]. Nonetheless, such a conception of function is definitely machinic, consisting of airflow, heat exchange or drainage, expressing a particular feeling of progress and achievement through industrial manufacturing techniques allowing for new material

[30] *Esthétique de l'ingénieur* is the title of one of the chapters of Le Corbusier's manifesto, *Vers une Architecture* (Le Corbusier & Saugnier, 1923)

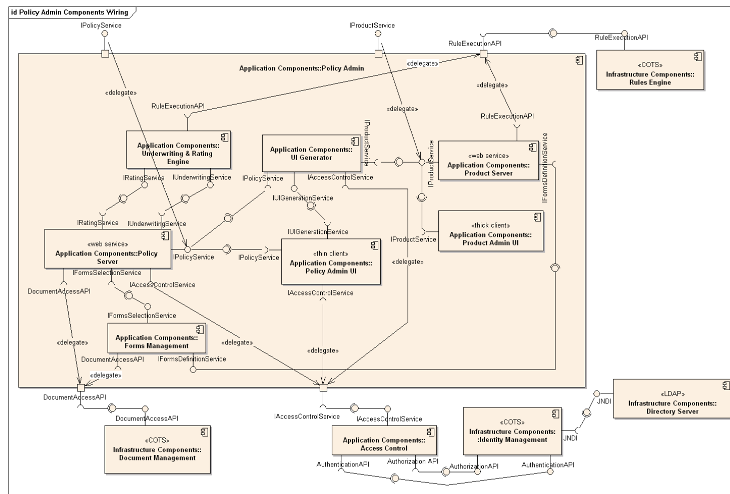

Figure 4.1: Description of a software component and its inner relations in the Universal Modelling Language, (Wikipedia, 2023b)

capabilities against contextual understandings. Here again, the human is but a small part in a dynamic system.

Jacques Rancière, in his study of the Werkbund and the Bauhaus-inspired architecture, offers an alternative approach, away from the strcit functionality laid out by Sullivan and Le Corbusier before him. The simplification of forms and processes, he writes of the AEG Turbinenhalle in Berlin, which is normally associated with the reign of the machine, finds itself, on the contrary, related to art, the only thing able to spiritualize industrial work and common life (Ranciere, 2013).

By paying attention to the role of a detail, and of the human subjectivity and situatedness of the people inhabiting the building, departs form the strict function of an object or of a building, to its actual use. Such a shift moves the aesthetic judgment from a structure-centric perspective (such as Le Corbusier's ideal dimensions), to a human-centric perspective (such as Lacaton & Vassal's practical extension of space and light). Peter Downton reiterates this point, when he states that "*buildings and design are often judged from artistic perspectives that bear no relation to how the building's*

occupants perceive or occupy the building." (Downton, 1998); his conception of the artistic here, is one that aligns with Kant's definition of a work that is purposive in itself, and not based on a function that it should fulfill.

One can see a translation of such a self-referential conception of art in the class of building which encompass follies and pavillions. These kinds of buildings are constructed first and foremost for their decorative properties, and only secondarily for its structural and functional properties. Follies, for instance, are individual buildings built on the demand of one specific individual's desire. They aim to represent something else than what they are, with no other purpose than ornament and the display of wealth. Pavilions, in the modern acceptation of the term, are rather displays of architectural and engineer prowess, demonstrating the use of new techniques and materials. By focusing only on design and technical feat, it is this prowess itself that is being represented: the function of the building is only to represent the skill of its builders. For instance, Junya Ishigami's pavillion at the Venice Biennale in 2008, shown in 4.2 consisted in a very elegant and aerial structure, but whose function was depending on the fact that no living being interacted with it[31].

As an artform, architecture provides an immersive and systemic physical environment, and thus shapes human psychology and agency within it, in turn forcing the dweller to acknowledge and engage with their environment. This suggests that, from a formal, top-down approach which considers architecture as possessing a systematic language to be realized exactly at a structural level, there exists a complementary, bottom-up approach, centered around human construction and function.

---

[31]Indeed, the structure collapsed due to a cat's playfulness: *"The Barbican says that the 37-year-old Ishigami is "internationally acclaimed", and there is certainly a buzz and fascination around him. Last year he won the Golden Lion, the highest prize at the Venice Architecture Biennale, for a structure that collapsed almost as soon as it was built, following an accident with a cat. Little was left but a scrawled note saying "Scusate, si è rotto. I'm sorry It's broken."* (Moore, 2011)"

Figure 4.2: Pavillion built by Junya Ishigami + associates, showing a focus on appearance and structural feat, rather than habitability. Picture courtesy of Iwan Baan, 2008.

4.3.2 Patterns and structures

A counterpoint to this modernist approach of master planning is that of Christopher Alexander. Along with other city planners in the United States, such as William H. Whythe or Jane Jacobs, Alexander belongs to an empirical tradition of determining what makes a built environment *good* or not, by examining its uses and the feelings it elicits in the people who tread its grounds. He elaborates an approach to architecture which does not exclusively rely on abstract design and technological efficiency, but rather takes into account the multiple layers and factors that go into making

> *[...] beautiful places, places where you feel yourself, places where you feel alive (Alexander, 1979) [...]*

In *The Timeless Way of Building*, he focuses on how beauty is involved in moving from disorganized to organized complexity, a design process which is not, in itself, the essence of beauty, but rather the condition

for such beauty to arise. Alexander's conception of beauty, while very present throughout his work, is however not immediately concerned with the specifics of aesthetics, but rather with the existence of such objects. This existence, in turn, does require to be experienced sensually, including visually.

In this process of achieving organized complexity, he highlights the paradoxical interplay between symmetry and asymmetry, and pinpoints beauty as the "deep interlock and ambiguity" of the two, a beauty he also finds the the relationship between static structures of the built environment, and the flow of living individuals in their midst. In his perspective, then, architecture should take into account the role of tension between opposite elements, rather than the combination of rational and abstract design elements. Such an approach echoes other considerations of tension as a source for stimulating human engagement,such as Ricoeur's analysis of the metaphor (see 4.2.1), and the resolution of the riddles presented in works of obfuscated source code (see 2.1.2).

He therefore considers a possible aesthetic experience as a consequence of qualities such as appropriateness, rightness to fit, not-simplistic and wholeness. All of these have in common the subsequent need for a purpose, a purpose which he calls the *Quality Without a Name* (Alexander, 1979). This quality, he says, is semantically elusive, but nonetheless exists; it is, ultimately, the quality which sustains life, a conclusion which he reached after extensive empirical research: no one can name it precisely, but everyone knows what it refers to. It is the quality which makes one feel at home, which makes one feel like things make sense in a deep, unexplicable way[32]. This reluctance to being linguistically explicited is echoed in

[32]"*It is always looking at two entities of some kind and comparing them as to which one has more life. It appears to be a rank bit of subjectivity. [...] It turns out that these kind of measurements do correlate with real structural features in the thing and with the presence of life in the thing measured by other methods, so that it isn't just some sort of subjected I groove to this, and I don't groove to that and so on. But it is a way of measuring a real deep condition in the particular things that are being compared or looked at.*" (Alexander, 1996)

the work of the craftsman, where a practitioner often finds herself showing rather than telling (Pye, 2008), another domain with which software developers identify, explicited in 2.3.3.

Among the adjectives used to circle around this quality are whole, comfortable, free, exact, egoless, eternal (Alexander, 1979). Some of these qualities can also be found in software development, particularly wholeness and comfort. A whole program is a program which is not missing any features, whose encounter (or lack thereof) might cause a crash. If if a function implies a systematic design, such systematic design is not compromised by the lacking of some parts. Conversely, it is also a program which does not have extraneous—useless—features.

A comfortable program text being is a program which might be modified without fear of some unintended side-effects, without inivisible dependencies which might then compromise the whole. There is enough separation of concerns to ensure a somewhat safe working area, in which one can engage in epistemic probing assuming that things will not be breaking in unexpected ways; being whole, it also provides a higher sense of meaning by realizing how one's work relates to the rest of the construction. The implication here is that comfort derives from a certain kind of knowledge, a knowledge of how things (spatial arrangements, technical specifications, human functions) are arranged, how they relate to each other, how they can be used and modified.

To complement this theoretical pendant, Alexander conducted empirical research to find examples of such qualities, in a study led at the University of Berkley which resulted in his most popular book, *A Pattern Language* (Alexander et al., 1977). In it, he and his team lists 253 patterns which are presented as to form a kind of language, akin to a Chomskian generative grammar, re-usable and extendable in a very concrete way, but without a normative, quasi-biological component. It turns it out that such a documentation, of re-usable configuration and solutions for contextual problem-solving, had a significant echo with computer scientists.

A whole field of research developed around the idea expressed in *A Pattern Language*, at the crossover between computer science and architecture[33] of distinct, self-contained but nevertheless composable components. In Alexandrian terms, they are a triad, which expresses a relation between a certain context, a problem, and a solution. Similarly to architectural patterns, these emerged in a bottom-up fashion: individual software developers found that particular ways of writing and organizing code were in fact extensible and reusable solutions to common problems which could be formalized enough to be shared with others. Patterns enable a cognitive engagement based on a feeling of familiarity, and of recognizing affordances.

Extending from the similarities of structure and function between software and architecture mentioned above, it is the lack of learning from practical successes and failures in the field which prompted interest in Alexander's work, along with the development of Object-Oriented Programming, first through the Smalltalk language[34], then with C++, until today, as most of the programming languages in 2023 include some sort of object-orientation and encapsulation. What object-orientation does, is that it provides a semantic structure to the program, reflected in the syntactic structure: objects are conceptual entities, with states and actions, as discussed in 3.2.2 and shown in 45. This enables such objects to be re-used within a program text, and even across program texts.

The similarities between a pattern and an object, insofar as they are self-contained solutions to contextual situations that emerged through practice, and resulting from empirical deductions, caught on with software developers as a technical solution with a social inflection, rather than a computational focus. Writing in *Patterns of Software*, with a foreword by

[33]See, for instance, the *Beautiful Software Initiative* as an organized effort to develop Alexander's theses on growth, order, artefact and computation (Bryant, 2022).

[34]For an extensive history of the design and development of the Smalltalk hardware and software, see Alan Kay's *Early History of Smalltalk* (Kay, 1993).

Alexander, Richard P. Gabriel addresses this shift from the machine to the human:

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

The real issue raised here in programming seems to be, again, not to speak to the machine, but to speak to other humans. The programming paradigm of object-orientation aims at solving such complexity in communication. While understanding software is hard, creating, identifying, and formalizing patterns into re-usable solutions turns out to be at least as hard (Taylor, 2001). Part of this comes from a lack of visibility of code bases (most of them being closed source), but also from the series of various economic and time-sensitive constraints to which developers are subject to (and echoes those in the field of architecture), and which result in moving from making something great to making something good enough to ship. The promise of software patterns seemed to offer a way out by—laboriously—codifying know-how. Interestingly, while the increase in software quality has been found to result from the application of engineering practices (Hoare, 1996), the discovery and formalization of the software patterns takes place through the format of writers' workshops[35],

[35] As taken from the website of the 2022 Pattern Languages of Programming conference: *"At PLoP, we focus on improving the written expression of patterns through writers' workshops. You will have opportunities to refine and extend your patterns with the assistance of knowledgeable*

presenting a different mode of knowledge transmission.

Throughout his work, Gabriel draws from the work of an architect to weave parallels between his experience as a software developer and as a poetry writer, drawing concepts from the latter field into the former, and inspecting it through the lens of a pattern languages of built concrete or abstract structures. We develop further two concepts in particular, and show how *habitability* and *compression* enable an understanding of such structures.

Compression and habitability in functional structures

We have seen how source code is an inherently spatial medium, with entrypoints, extracted packages, parallel threads of executions, relative folders and directories and endless jump between files. Reading a program text therefore matches more closely an excursion into a foreign territory whose map might be misleading, than reading a book from start to finish. For instance, 4.3 builds on a longer history of using the city as a metaphor for large code bases, and visualizes classes, packages and version in three dimensions.

Given this somewhat literal mapping of source code structure onto urban structure, and given the abstract structure of object-oriented code, a reader of source code will need to find their bearings and orient themselves[36]. Once the entrypoint is found, the programmer starts to explore the programmed maze and attempts to make sense of their surroundings, as a step towards the construction of mental models.

Both inhabitants in a building and programmers in a code base have a tendency to be there to *accomplish something*, whether it might be living,

and sympathetic patterns enthusiasts and to work with others to develop pattern languages." (Guerra & Manns, 2022).

[36]"*Exploring a source code repository always starts with finding out what the OS will select as the entry point. 99% of the time it means finding the 'int main(int,char**)' function*" says Fabien Sanglard on the topic of reverse-engineering code-bases (Sanglard, 2018).

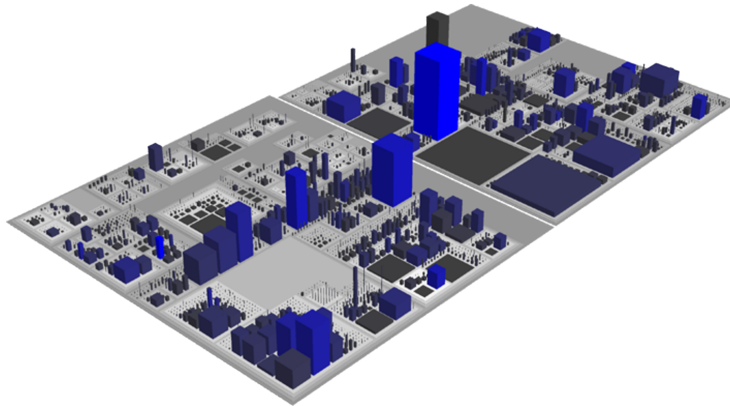

Figure 4.3: CodeCity is an integrated environment for software analysis, in which software systems are visualized as interactive, navigable 3D cities. The classes are represented as buildings in the city, while the packages are depicted as the districts in which the buildings reside. (Wettel, 2008)

working or eating for the former, or fixing, learning or modifying for the latter.s Particularly in software, one of the correlated functions of a program text is to be maintainable; that is, it must be made under the assumption that others will want to modify and extend source code. Other pieces of code might just be satisfying in being read or deciphered (as we've seen in source code poetry in 2.3.1 or with hackers in 2.1.2) but this assumption of interaction with the code brings in another concept, that of *habitability*. In Gabriel's terms, it is

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

In a sense, then, beautiful code is also code that is clear enough to inform action and, well-organized enough to warrant actually taking that action. For instance, writing in the ACM Queue, an anonymous programmer discusses the beauty in a code where the separation between which

sections of the source are hardware-dependent and which are not, as seen in 54. In that example, it is clear to the programmer what the problem-domain is: counter incrementation, high-performance computation, or a specific Intel chip.

There are several things which we can identify here. First, the three lines at the top of the listing indicate version numbers, which do not hold any computational functionality, but rather a human functionality: it communicates that this software considers change and evolution as core part of its source code, inviting the programmer reader to further modify it[37]

Second, the line defining the types of CPUs supported by the software is written in human-intelligible way, rather than a cryptic hexadecimal notation[38]. While the CPUs are ultimately represented in hexadecimal notation, the effort is made to render things intelligible to and quickly retrievable from the programmer's memory.

Finally, the `struct pmc_mdep` is a shorthand notation for "machine-dependent". In an era in which software can theoretically be executed on different hardware architectures, it is welcome to make the difference between the variables themselves, which apply across platform, and the values of these variables, which need to be changed per platform[39]. This is

---

[37]From the anonymous programmer: *"The engineer clearly knew his software would be modified not only by himself but also by others, and he has specifically allowed for that by having major, minor, and patch version numbers. Simple? Yes. Found often? No."* (Vicious, 2008).

[38]*"Nothing is more frustrating when working on a piece of software than having to remember yet another stupid, usually hex, constant. I am not impressed by programmers who can remember they numbered things from 0x100 and that 0x105 happens to be significant. Who cares? I don't. What I want is code that uses descriptive names. Also note the constants in the code aren't very long, but are just long enough to make it easy to know in the code which chip we're talking about."* (Vicious, 2008).

[39]*"It would seem obvious that you want to separate the bits of data that are specific to a certain type of CPU or device from data that is independent, but what seems obvious is rarely done in practice. The fact that the engineer thought about which bits should go where indicates a high level of quality in the code."* (Vicious, 2008).

```
#define        PMC_VERSION_MAJOR        0x03
#define        PMC_VERSION_MINOR        0x00
#define        PMC_VERSION_PATCH        0x0000

/* * Kinds of CPUs known */

#define __PMC_CPUS() \ __PMC_CPU(AMD_K7, "AMD K7") \
↪  __PMC_CPU(AMD_K8, "AMD K8") \ __PMC_CPU(INTEL_P5, "Intel
↪  Pentium")        \ __PMC_CPU(INTEL_P6, "Intel Pentium Pro") \
↪  __PMC_CPU(INTEL_CL, "Intel Celeron") \ __PMC_CPU(INTEL_PII,
↪  "Intel Pentium II") \ __PMC_CPU(INTEL_PIII, "Intel Pentium III")
↪  \ __PMC_CPU(INTEL_PM, "Intel Pentium M") \ __PMC_CPU(INTEL_PIV,
↪  "Intel Pentium IV")

// ...

/*
 * struct pmc_mdep
 *
 * Machine dependent bits needed per CPU type.
 */

struct pmc_mdep
{
    uint32_t pmd_cputtype; /* from enum pmc_cputype */
    uint32_t pmd_npmc;     /* max PMXs per CPU */
    uint32_t pmd_npmc;     /* PMC classes supported */
    struct pmc_classinfo pmd_classes[PMC_CLASS_MAX];
    int pmd_nclasspmcs[PMC_CLASS_MAX];

    /*
     * Methods
     */

    int (*pmd_init)(int _cpu);    /* machine dependent initalization*/
    int (*pmd_cleanup)(int _cpu) /* machine dependent cleanup */
}
```

Listing 54: This header file defines the structure of a program, both in its human use, in its interaction with hardware components, and its decoupling of hardware and software elements.

a good example of a separation of concerns: it is made clear which parts of the program text the programmer needs to pay attention to, and can change, and which parts of the program texts she needs not be concerned with. For a further example of separation of concerns, one could point a beautiful commit is a commit which adds a significant feature, and yet only change the lines of the code that are within well-defined boundaries (e.g. a single function), leaving the rest of the codebase untouched, and yet affecting it in a fundamental way.

Habitability, then, is a combination of acknowledgment by the writer(s) to the reader(s) of the source, by referring to the evolution over time of the software, along with the use of intelligible names and separation of concerns. This distinction relates to Alexander's property of comfort, by affording involvement instead of estrangement. Still, such a feature of habitability, of supporting life, doesn't specify at all what it could, or should, look like. Rather, we get from Alexander a negative definition:

> *The details of a building cannot be made alive when they are made from modular parts...And for the same reason, the details of a building cannot be made alive when they are drawn at a drawing board. (Alexander, 1979)*

If modularity itself is at odds with making good (software) constructions, then its implementation under the terms of an object-oriented programming paradigm becomes complicated. Indeed, the technical formalization of the field came with the release of the *Design Patterns: Elements of Reusable Object-Oriented Software* book, which lists 23 design patterns implementable in software (Gamma et al., 1994). Its influence, in terms of copies sold, and in terms of papers, conferences and working groups created in its wake, is undeniable, with Alexander himself giving a keynote address at the ACM two years after the release. It has, however, been met with some criticism.

Some of this criticism is that patterns are "external", they look like

they come from somewhere else, and are not adapted to the code. In this sense, this corroborates Alexander in being wary of constructions which do not integrate fully within their environments, which do not, in an organic sense, allow for a piecemeal growth[40]. If patterns express relations between contexts, problems and solutions, then it seems that one of the main complaints of developers is that they might, one day, look at the code they were working on and see chunks of foreign snippets dumped in the middle to fix some generic problem, with no understanding for specifics, nor fit in the existing structure. This is judged negatively due to its lack of understanding of context offered by those proposed solutions. In this, blindly applying patterns from a textbook might be a solution, but it's not an elegant one. This criticism also finds its echoes in the Scruton's analysis of architectural styles; rules and conventions, while present in architecture, are often adopted only to be departed from—re-interpreted and adapted to the context of the building (Scruton, 2013).

One aspect that has been eluded so far is therefore that of the programming languages used by the programmer. Indeed, one doesn't write Ruby like one writes Java, C++, or Lisp. If materiality is a core component of eliciting an aesthetic experience in an architectural context, then programming languages are the material of source code, and offer a specific context to the writing and reading of the program text.

A final criticism to software patterns is that they are language-independent. As such, they are often workarounds for features that a particular programming language doesn't allow from the get-go, or offers simpler implementations than the pattern's[41].

While patterns might operate at a more structural level, hinting at different parts of code, and its overall organization, one can also turn to a

[40] Addressing this concern, the failure of strict top-down hierarchies in software development resulted in the *agile* methodology for business teams, now one of the most popular ways of building software products.

[41] For instance, Peter Norvig highlights that most patterns in the original book have much simpler implementations in Lisp than in C++ or Smalltalk (Norvig, 1998)

more micro-level. What can a detail do in our understanding of structures? Sometimes decried, sometimes praised in architecture, the detail fulfills mutliple roles: acting as a meaningful interface, compressing meaning and testifying for materiality.

Both Scruton and Rancière mention the detail as an essential architectural element. Without contributing to the structural soundness of the construction, it nonetheless contributes to its expressiveness. A blend of the cognitive and sensual is also characteristic of Scruton's "imaginative perception", understood as the perception of the details of built structures, and their extrapolation into the imaginary. Indeed, the experience of the user is based on the points at which it sensually grasps its environment: the detail is therefore the point of interaction between the human and the structure. This imagination depends on the interpretative choices in parsing ambiguous or multiform aspects of the built environment. The detail contributes to the stylistic convention of the creation:

> *Convention, by limiting choice, makes it possible to 'read' the meaning in the choices that are made ...for style is used to 'root' the meanings which are suggested to the aesthetic understanding, to attach them to the appearance from which they are derived. (Scruton, 2013)*

With many external constraints, due to both context and function, the architect or builder does not have much room for personal expression, and it is through details that their intent and their style are being shown. The significance of a detail can be in explaining which conventions the structure adopts, as well as communicating the intent of the creator. A significant detail manages to compress meaning into a restricted physical surface.

Compression is a concept introduced by Gabriel in response to pattern design. In narrative and poetic text, it is the process through which a word is given additional meaning by the rest of the sentence. In a sentence such

as *"Last night I dreamt I went to Manderley again."* (Du Maurier, 1938), the reader is unlikely to be familiar with the exact meaning of *Manderley*, since this is the first sentence of the novel. However, we can infer some of the properties of Manderley from the rest of the sentence: it is most likely a place, and it most likely had something to do with the narrator's past, since it is being returned to. A similar phenomenon happens in source code, in which the meaning of a particular expression or statement can be derived from itself, or from a larger context. In object-oriented programming, the process of inheritance across classes allows for the meaning of a particular subclass to be mostly defined in terms of the fields and methods of its subclasses—its meaning is compressed by relying on a semantic environment, which might or not be immediately visible.

This, Gabriel says, induces a tension between extendability (to create a new subclass, one must only extend the parent, and only add the differentiating aspects) and context-awareness (one has to keep in mind the whole chain of properties in order to know exactly what the definition of an interface that is being extended really is). Resolving such a tension, by including enough information to hint at the context, while not over-reaching into idiosyncracy, is a thin line of being self-explanatory without being verbose.

For instance, Casey Muratori explores the process of compression in refactoring a program text, first by distinguishing semantic compression from syntactic compression[42], and then honing in on what makes a compression successful[43]. Transitioning from uncompressed code, shown in 55 to compressed code, shown in56, allows the programmer to understand

---

[42]*"Like, literally, pretend you were a really great version of PKZip, running continuously on your code, looking for ways to make it (semantically) smaller. And just to be clear, I mean semantically smaller, as in less duplicated or similar code, not physically smaller, as in less text, although the two often go hand-in-hand."* (Muratori, 2014)

[43]*"Ah! It's like a breath of fresh air compared to the original, isn't it? Look at how nice that looks! It's getting close to the minimum amount of information necessary to actually define the unique UI of the movement panel, which is how we know we're doing a good job of compressing.* (Muratori, 2014)

```
int num_categories = 4;
int category_height = ypad + 1.2 * body_font->character_height;
float x0 = x;
float y0 = y;
float title_height = draw_title(x0, y0, title);
float height = title_height + num_categories * category_height + ypad;
my_height = height;
y0 -= title_height;

{
    y0 -= category_height;
    char *string = "Auto Snap";
    bool pressed = draw_big_text_button(x0, y0, my_width,
    ↪  category_height, string);
    if (pressed)
        do_auto_snap(this);
}

{
    y0 -= category_height;
    char *string = "Reset Orientation";
    bool pressed = draw_big_text_button(x0, y0, my_width,
    ↪  category_height, string);
    if (pressed)
    {
        // ...
    }
}
// ...
```

Listing 55: genalloc.c, Basic general purpose allocator for managing special purpose memory from the Linux Kernel, displaying examples of source-code spatiality (Muratori, 2014)

broad patterns about the overall architecture of the program text—here, the function is to display a clickable panel on a user interface.

The difference we can see between the compressed and uncompressed goes beyond the number of lines used for the same functionality. A first clue in terms of semantics is the use of strictly syntactic block markers: { and }. There are here stricly to delimitate a code block, with no semantic meaning to the computer. While the uncompressed listing shows all the separate elements needed for a button to exist (such as `x0`, `y0`, `my_height`, etc.), while the compressed listings as encapsulated them into an object

```
Panel_Layout layout(this, x, y, my_width);
layout.window_title(title);

layout.row();
if(layout.push_button("Auto Snap")) {
    do_auto_snap(this);
}

layout.row();
if(layout.push_button("Reset Orientation"))
{
    // ...
}

// ...
layout.complete(this);
```

Listing 56: genalloc.c, Basic general purpose allocator for managing special purpose memory from the Linux Kernel, displaying examples of source-code spatiality (Muratori, 2014)

called `Panel_Layout`, thus abstracting away from the programmer's mind the details of such a panel. This encapsulation then enables a further compression of the program: by adding the `push_button()` method on the `layout`, the compressed code realizes the same functionality of checking for button presses for each button, but ties it to a specific object and, due to the implementation, includes the name of the button being pressed on the same line as the check happens, rather than a line apart in the uncompressed example.

By compressing the source code and abstracting some concepts, such as the button, one can also gain understanding about the rest of the program text. By showing details of practices and styles, a programmer can extrapolate from a small fragment to a larger structure. Gabriel calls this idea *locality*: it is

> *that characteristic of source code that enables a programmer to understand that source by looking at only a small portion of it. (Gabriel, 1998)*

In poetry, compression presents a different problem since, ultimately, the definitions of each words are not limited to the poet's own mind but also exist in the broad conceptual structures which readers hold. However, since all aspects of a program is always explicitly defined, programmers thus have the ultimate say on the definition of most of the data and functions described in code. As such, they create their own semantic contexts while, at the same time, having to take into account the context of the machine, the context of the problem, and the context(s) that their reader(s) might be coming from.

We now see that, within the same need for the appreciation of function, architecture can take opposite approaches: seeing a building as an abstract design, or as a concrete construction. In his 1951 lecture, "Building, Dwelling, Thinking", Martin Heidegger offers a perspective on these two forms of architecture. He equates top-down and bottom-up to, respectively, building as erecting, and building as cultivating. Ultimately, both of these approaches relate to human dwelling in a given location. To dwell is an engagement of thought and of action, one which leads to the construction of buildings in particular locations, arguing for a contextual adequacy of human structures to their environment[44] (Heidegger & Hofstadter, 1975). This active existence in time and space, allowing for deliberate thought and action and resulting in a better structure also equates to Gabriel's concept of habitability:

> *Habitability is the characteristic of source code that enables programmers coming to the code later in its life to understand its construction and intentions and to change it comfortably and confidently ...Software needs to be habitable because it always has to change ...What is important is that it be easy for programmers to come up to speed with the code, to be able to navigate*

[44] Speaking of a farmhouse in the Schwarzwald, he describes the chain of creation as such: *" A craft which, itself sprung from dwelling, still uses its tools and frames as things, built the farmhouse.*

> *through it effectively, to be able to understand what changes to make, and to be able to make them safely and correctly. (Gabriel, 1998)*

As Heidegger returns to the etymological root of dwelling (*bauern*) in order to connect it to the possible experience of the world humans can have through language, he grounds our experience in context. His though, between earth, man, *techne* and construction, hints at the essence that human construction—craft—as a consequence of thought and as a precedence to construction. Taking into account context and materiality, a final connection between software and architecture is actually with the field that predated, and complemented, architecture: craftsmanship.

4.3.3 Material knowledge

Architecture as a field and the architect as a role have been solidified during the Renaissance, consecrating a separation of abstract design and concrete work. This shift obfuscates the figure of the craftsman, who is relegated to the role of executioner, until the arrival of civil engineering and blueprints overwhelmingly formalized the discipline (Pevsner, 1942). While computer science, through its abstract designs, echoes the modernist architect with its pure plans, the programmer, identifying itself with the craftsman, offers different avenues for knowing artefacts.

The architect emerged from centuries of hands-on work, while the computer scientist (formerly known as mathematician in the 1940s and 1950s) was first to a whole field of practitioners as programmers, followed by a need to regulate and structure those practices. Different sequences of events, perhaps, but nonetheless mirroring each other. On one side, construction work without an explicit architect, under the supervision of bishops and clerks, did indeed result in significant achievement, such as Notre Dame de Paris or the Sienna Cathedral. On the other side, letting go of structured and restricted modes of working characterizing computer pro-

gramming up to the 1980s resulted in a comparison described in the aptly-named *The Cathedral and the Bazaar*. This essay described the Linux project, the open-source philosophy it propelled into the limelight, and how the quantity of self-motivated workers without rigid working structures (which is not to say without clear designs) can result in better work than if made by a few, select, highly-skilled individuals (Raymond, 2001; Henningsen & Larsen, 2020).

What we see, then, is a similar result: individuals can cooperate on a long-term basis out of intrinsic motivation, and without clear, individual ownership of the result; a parallel seen in the similar concepts of *collective craftsmanship* in the Middle-Ages and the *egoless programming* of today (Brooks Jr, 1975). Building complex structures through horizontal networks and practical knowledge is therefore possible, with consequences in terms aesthetic apprecitations.

Craftsmanship in our contemporary discourse seems most tied to a retrospective approach: it is often qualified as that which was *before* manufacture, and the mechanical automation of production (Thompson, 1934), preferring materials and context to technological automation. Following Sennett's work on craftsmanship as a cultural practice, we will use his definition of craftsmanship as *hand-held, tool-based, intrinsically-motivated work which produces functional artefacts, and in the process of which is held the possibility for unique mistakes* (Sennett, 2009).

At the heart of knowledge transmission and acquisition of the craftsman stands the *practice*, and inherent in the practice is the *good practice*, the one leading to a beautiful result. The existence of an aesthetic experience of code, and the adjectives used to qualify it (smelly, spaghetti, muddy), pointed at in 2.2.2, already hints at an appreciation of code beyond its formalisms and rationalisms, and towards its materiality.

A traditional perspective is that motor skills, with dexterity, care and experience, are an essential feature of a craftsman's ability to realize something beautiful (Osborne, 1977), along with self-assigned standards of qual-

ity (Pye, 2008; Sennett, 2009). These qualitative standards which, when pushed to their extreme, result in a craftsperson's *style*, gained through practice and experience, rather than by explicit measurements (Pye, 2008) [45]. Two things are concerned here, supporting the final result: tools and materials (Pye, 2008). According to Pye, a craftsperson should have a deep, implicit knowledge of both, what they use to manipulate (chisels, hammers, ovens, etc.) as well as what they manipulate (stone, wood, steel, etc).

The knowledge that the craftsman derives, while being tacit (see 3.1.2), is directed at its tools, its materials, and the function ascribed to the artefact being constructed, and such knowledge is derived from a direct engagement with the first two, and a constant relation to the third. Finally, any aesthetic decoration is here to attest to the care and engagement of the individual in what is being constructed—its dwelling, in Heideggerian terms.

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

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

He opens here up a solution to the paradox of the hand-made and the computer-automated, as programming emerges from the latter as a new skill. This very rise of automation has been criticized for the rise of a Osborne's "soulless society" (Osborne, 1977), and has triggered debates about authorship, creativity and humanity at the cross-roads between artificial intelligence and artistic practice (Mazzone & Elgammal, 2019). One av-

[45] See Pye's account of craftsmanship, and his intent to make explicit the question of quality craftsmanship and *"answer factually rather than with a series of emotive noises such as protagonists of craftsmanship have too often made instead of answering it."* (Pye, 2008)

enue out of this debate is human-machine cooperation, first envisioned by Licklider and proposed throughout the development of Human-Computer Interaction (Licklider, 1960; Grudin, 2016). If machines, more and more driven by computing systems, have replaced traditional craftsmanship's skills and dexterity, this replacement can nonetheless suggest programming as a distinctly 21st-century craftsmanship, as well as other forms of cratsmanship-based work in an information economy.

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

A craftsman's material consciousness is recognized by the anthropomorphic qualities ascribed by the craftsman to the material (Sennett, 2009), the personalities and qualities that are being ascribed to it beyond the immediate one it posseses. Clean code, elegant code, are indicators not just of the awareness of code as a raw material that should be worked on, but also of the necessities for code to exist in a social world, echoing Scruton's analysis that architectural aesthetics cannot be decoupled from a social sense[46]. As software craftsmen assemble in loose hierarchies to construct software, the aesthetic standard is *the respect of others*, as mentioned in computer science textbooks (Abelson et al., 1979).

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

[46]"*it is the aesthetic sense which can transform the architetct's task from the blind pursuit of an uncomprehended function into a true exercise of practical common sense.*" (Scruton, 2013)

with words[47].

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

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

So while there are arguments for developing a more rigorous, engineering conception of software development (Ensmenger, 2012), a crafts ethos based on a materiality of code holds some implications both for programmers and for society at large: engagement with code-as-material opens up possibilities for the acknowledgement of a different moral standard[48]. As Pye puts it,

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

Code well-done is a display of excellence, in a discipline in which excellence has not been made explicit. If most commentators on the history of

---

[47] This disregards the impact of programming languages, the hardware they run on, and the data they process on the environment; see (Kurp, 2008)

[48] Writing about resilient web development, Jeremy Keith echoes this need for material honesty: *"The world of architecture has accrued its own set of design values over the years. One of those values is the principle of material honesty. One material should not be used as a substitute for another. Otherwise the end result is deceptive (Keith, 2016). "*

craftsmanship lament the disappearance of a better, long-gone way of doing things, before computers came to automate everything, locating software as a contemporary iteration of the age-old ethos of craftsmanship nonetheless situates it in a longer tradition of intuitive, concrete creation.

To conclude this section, we have seen that architecture can offer us some heuristics when looking for aesthetic features which code can exhibit. Starting from the naïve understanding that form should follow function, we've examined how Alexander's theory of patterns, and its significant influence on the programming community[49], points not just to an explicit conditioning of form to its function (in which case we would all write hand-made Assembly code), but rather to an elusive, yet present quality, which is both problem- and context-dependent.

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

4.4 Forms of scientific activity

As programmers learned their craft from practice and immediate engagement with their material, computer science was concomittantly develop-

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

ing from a seemingly more abstract discipline. Mathematicians such as Alan Turing, John Von Neumann and Grace Hopper can be seen, not just as the foreparents of the discipline of computing, but also as standing on the shoulders of a long tradition of mathematicians. Computation is one of the many branches of contemporary mathematics and, as it turns out, this discipline also has reccuringly included references to aesthetics. After the metaphors of literature and the patterned structures of architecture, we conclude our analysis of the aesthetic relation of domains contingent to source code by looking at how mathematics integrate formal presentation.

This section approaches the topic of aesthetics and mathematics in three different steps. First, we look at the objective or status of beauty in mathematics: are mathematical objects eliciting an aesthetic experience in and of themselves, or do they rely on the observer's perception? Considering the difference between abstract objects and their representation: is aesthetic representation ascribed to either, or to both? And what is the place of the observer in this judgment? Having established a particular focus on the representations of abstract objects, we then turn to the epistemic value of aesthetics, and how positive aesthetic representations in mathematics can enable insight and understanding. Finally, we complement this relation between knowledge and presentation and depart from the ends of a proof, and an evaluative appraisal of aesthetics in mathematics, by investigating the actual process of doing mathematics, concluding with topics of pedagogy and ethics.

4.4.1 Beauty in mathematics

The object of mathematics is to deal first and foremost with abstract entities, such as the circle, the number zero or the derivative, which can find their applications in fields like engineering, physics or computer science. Because of this historical separation from the practical world through

the use and development of symbols, one of the dominant discourses in the field tended to consider mathematical beauty as something intrisic to itself, and independent from time, culture, observer, or representation itself. Indeed, a circle remains a circle in any culture, and its aesthetic properties—uniformity, symmetry—do not, at first glance, seem to be changing across time or space.

According to the Western tradition, mathematics are perhaps the first art. Aristotle, in his *Metaphysics*, wrote of beauty and mathematics as the former being most purely represented by the latter, through properties such as order, symmetry and definiteness[50]. By offering insight into the harmonious arrangement of parts, it was thought that mathematics could, through beauty, provide knowledge of the nature of things, resulting in an understanding of how things generally fit together. Beauty then naturally emerges from mathematics, and mathematics can, in turn, provide an example of beauty. At this intersection, it also becomes a source of intellectual pleasure, since gaining mathematical knowledge exercises the human being's best power—that of the mind.

Arguing for this position of objective quality being revealed through beautiful manifestation, Godfrey H. Hardy writes, in his *Mathematician's Apology*, that beauty is constitutive of the objects that compose the field; their abstract quality is what removes them from the contextuality of human judgment.

> *A mathematician, like a painter or a poet, is a maker of patterns. If his patterns are more permanent than theirs, it is because they are made with* ideas*. A painter makes patterns with shapes and colours, a poet with words. ...The mathematician's patterns, like the painter's or the poet's must be* beautiful*; the ideas like*

[50]"*the supreme forms of beauty are order, symmetry, and definiteness, which the mathematical sciences demonstrate in a special degree. And since these (e.g. order and definiteness) are obviously causes of many things, evidently these sciences must treat this sort of causative principles also (i.e. the beautiful) as in some sense a cause.*" (Aristotle, 2006)

the colours or the words, must fit together in a harmonious way. Beauty is the first test: there is no permanent place for ugly mathematics. (Hardy, 2012)

Here, Hartman posits that it is the arrangement of ideas that possess aesthetic value, and not the arrangement of the representation of ideas. In this, he follows the position of other influential mathematicians, such as Poincaré (Poincaré, 1908), or Dirac (Kragh, 2002), who rely on beauty as a property of the mathematical object in itself. For instance, Dirac states that a physical law must necessarily stem from a beautiful mathematical theory, thus asserting that the epistemic content of the theory and its aesthetic judgment thereof are inseparable; a good mathematical theory is therefore intrinsically beautiful. Summing up these positions, Carlo Cellucci establishes proportion, order, symmetry, definiteness, harmony, unexpectedness, inevitablity, economy, simplicity, specificity, and integrations as the different properties inherent to mathematical objects, as mentioned from an essentialist perspective (Cellucci, 2015). Ironically, this rather seems to hint at the multiplicity of appreciations of beauty within mathematics, with mathematicians concurring on the existence of beauty, but not agreeing on what kind of beauty pertains to mathematics. Nonetheless, they do agree that beauty is connected to understanding and epistemic acquisition. John Von Neumann, writing in 1947, states that:

One expects a mathematical theorem or a mathematical theory not only to describe and to classify in a simple and elegant way numerous and a priori disparate special cases. One also expects "'elegance" in its "architectural," structural makeup. Ease in stating the problem, great difficulty in getting hold of it and in all attempts at approaching it, then again some very surprising twist by which the approach, or some part of the approach, becomes easy, etc...(Von Neumann, 1947)

The point that Von Neuman makes here is a difference between the con-

tent of the mathematical object and its structural form. Such a structural form, by organizing the connection of separate parts into a meaningful whole, makes it easy to grasp the problem. In this sense, it is both the crux of aesthetics and the crux of understanding.

Similarly, François Le Lionnais, a founding member of the Oulipo literary movement in postwar France, wrote an essay on the aesthetic of mathematics, paying attention to both the mathematical objects in and of themselves, such as e or π, but also to mathematical methods, and how they compare to traditional artistic domains such as classicism or romanticism. Without getting into the intricacies of this argumentation, we can nonetheless note that his descriptions of mathematical beauty find echoes in source code beauty. For instance, his appraisal of the proof by recurrence[51] reflects similar lines of praise given by programmers to the elegance of recursive functions, which are sharing the same mathematical device (for instance, see 32 and 17 for examples of recursion as an aesthetic property). A proof by recurrence is indeed a kind of structure, which can be adapted to demonstrate different kinds of mathematical objects.

To understand is to grasp how each elements fits with others within a greater structure (either in a poem, a symphony or a theorem), with some or all of these elements being rendered sensible to the observer (Cellucci, 2015). The beauty of a mathematical object can then be ascribed in its display of the definite relation between its elements. For instance, the equation representing Euler's identity (see 4.4) demonstrates the relation between geometry, algebra and numerical analysis through a restrained set of syntactic symbols, where e is Euler's number, the base of natural logarithms, i is the imaginary unit, which by definition satisfies i2=-1, and π is

[51]"*It seems to us that a method earns the epithet of classic when it permits the attainment of powerful effects by moderate means. A proof by recurrence is one such method. What wonderful power this procedure possesses! In one leap it can move to the end of a chain of conclusions composed of an infinite number of links, with the same ease and the same infailliability as would enter into deriving the conclusion in a trite three-part syllogism.*"" (Le Lionnais, 1971)

$$e^{i\pi} + 1 = 0$$

Figure 4.4: Euler's identity demonstrates the relation between geometry, algebra and numerical analysis through a restrained set of syntactic symbols.

the ratio of the circumference of a circle to its diameter. Each of the symbols is necessary, definite, and establishes clear relations between each other, revealing a deep interlock of simplicity within complexity.

There is also empirical grounding for such a statement. This equation ranked first in a column in the *Mathematician Intelligencer* about the beauty of mathematical objects; the columnist, David Wells, had asked readers to rank given theorems, on a linear scale from 0 to 10, according to how beautiful they were considered (Wells, 1990)[52]. Again, while this assessment does show that there can be consensus, and thus some aspect of objectivity, in a mathematician's judgment of beauty in a mathematical object, it also showed that mathematical beauty also depends on the observer, since mathematicians provided varying accounts.

Rather than focusing on the beauty of the mathematical entities them-

[52] Along with, for instance, the infinite prime theorem, and Fermat's "two squares" theorem.

selves, then, another perspective is to consider beauty to be found in the *representation* of mathematical , since conceptual entities can only graspable through their manifestation.

A first approach is to consider that that the beauty ascribed to mathematics and the beauty ascribed to mathematical representation are unrelated. This disjunctive view, that aesthetics and mathematics can be decoupled (e.g. there can be ugly proofs of insightful theorems, and elegant proofs of boring theorems), was first touched upon by Kant. As Starikova highlights, the philosopher operates a distinction between perceptual, disinterested beauty, and intellectual, vested beauty. Perceptual beauty, the one which can be found in the visual representations of mathematical entities, is the only beauty graspable, while intellectual beauty, that of the objects themselves, simply does not exist, "mathematics by itself being nothing but rules" (Starikova, 2018).

Such manifestation of perceptual beauty, connected to mathematical entities themselves, can nonetheless be found in the phenomenon of reproving in existing proofs, in order to make them more beautiful. Rota, for instance, associates the beauty of a piece of mathematics with the shortness of its proof, as well as with the knowledge of the existence of other, clumsier proofs[53] (Rota, 1997). Thus, it is not so much the content of the proof itself, nor the abstract mathematical object being proven that is the focus of aesthetic attention, but rather the process of establishing the epistemic validity of such an object.

What is useful here is technique, the demonstration of the knowledge from the prover to the observer, through the proof. As such, the asssessment of aesthetics in mathematics, both as a producer and as an observer, depends on the expertise of each individual, and on the previous knowl-

[53]"*"The beauty of a piece of mathematics is frequently associated with shortness of statement or of proof."* and *"A proof is viewed as beautiful only after one is made aware of previous, clumsier proofs."* (Rota, 1997)

edge that this indivual has of mathematics[54] (an assessment of the aesthetics of mathematics for non-expert is discussed in 4.4.3 below). It seems that the way that the mathematical object is presented does matter for the assessment.

If beauty is not intrinsic to the mathematical object, but rather connected to the representation of the mathematician's knowledge, there remains the question of why is beauty taken into account in the doing of mathematics. Looking at the lexical fields used by mathematicians to qualify their aesthetic experience, as reported in (Inglis & Aberdein, 2015) provides us with a clue: amongst the most used terms are "ingenious", "striking", "inspired", "creative", "beautiful", "profound" and "elegant". Some of these terms have a connection to the epistemic: for instance, something ingenious enables previously unseen connections between concepts, implying the resourcefulness and the cleverness of the originator of the idea. The next question is therefore that of the relationship between the aesthetic and the epistemic in mathematics; and in how this relation can manifest itself in source code.

4.4.2 Epistemic value of aesthetics

Caroline Jullien offers an alternative to the perception of mathematics as an autotelic aesthetic object, by retracing the definitions of beauty given by Aristotle and establishing a cognitive connection through a cross-reading of the *Metaphysics* and the *Poetics*, highlighting that "*the characteristics of beauty are thus useful properties that yield an optimal perception of the object they apply to. [...] Men can understand what is ordered, measured and delineated far better than what is chaotic, without clear boundaries, etc.*" (Jullien, 2012). She then develops this point further, building on Poincaré's

[54] "*Mathematical creation is not so free, hence the contrasting analogy of the landscape gardener, who needs a good grasp of the topography before getting down to creating something beautiful which needs to be based on that topography.*" (Thomas, 2017)

assessment of mathematical entities which fulfill aesthetic requirements and are, at the same time, an assistance towards understanding the whole of the mathematical object presented. Aesthetics, then, might not exist exclusively as intrinsic properties of a mathematical object, but rather as an epistemic device.

Her argument focuses on considering mathematics as a language of art in the Goodmanian sense of the term, investigating how mathematical notation relates to Goodman's criteria of syntactic density, semantic repleteness, semantic density, exemplification and multiple references (Jullien, 2012). She shows that, while mathematical notation might not seem to satisfy all criteria (for instance, syntactic density is only fulfilled if one takes into account graphical representations), a mathematical reasoning can present symptoms of the aesthetic, particularly through the ability to exemplify and refer to abstract entities.

However, to do that, she also includes different representations of mathematical systems, beyond typographical characters. Taking into account diagrams and graphs, it becomes easier to see how a more traditionally artistic representation of mathematics is possible. The thickness of a line, the color-coding or the spatial relationship can all express a particular class of mathematical objects; for instance, the commutative property in arithmetic can be represented in geometry through the aesthetic property of symmetry. In this work, we focus on the textual representation of source code, eluding any graph or diagram (such as the one we've seen in architectural descriptions of software systems in 4.1). Nonetheless, we have argued in 4.1 that source code qualifies as a language of art: while the syntactic repleteness does not match that of, say, painting, the unlimited typographical combinations, paired with the artificial design of programming languages as working medium enables the kinds of subtle distinctions necessary for symptoms of the aesthetic to be present.

Following Jullien, if a piece of mathematics is eliciting an aesthetic experience, or presenting positive aesthetic properties, it might then be a

support for the understanding by the viewer of this very piece of mathematics. Such a support is itself manifested in this ability to show a harmonious correspondence of parts in relation to a whole. A beautiful presentation is a cognitively encouraging presentation. The subsequent question then regards the nature of that understanding: if it does not happen as an instant stroke of enlightenment, how does it take place as a gradual process of deciphering (Rota, 1997)?

Addressing this question, Carlo Celluci hints at the concept of fitness, meaning the appropriateness of a symbol in its denotation of a concept, and the appropriateness of concepts in their demonstration of a theorem. Only through this dual level can fitness enable understanding rather than explanation (Cellucci, 2015). This gradual conception of understanding fits the context of proofs and demonstration; when confronted with source code—that is, with the result of a thought process of one or more programmers—the processual conception of understanding seems to find its limits.

To illustrate the relation between presentation and understanding of defined conceptual entities, we can look at how the linked list, a data structure that is fundamental in computer science, can be represented in an elegant way. The linked list allows for the retrieval and manipulation of connected items, as well as for the re-arrangement of the list itself. To do so, each item on the list contains both its value, and the address of the next item on the list, except for the last item, which points to `null`; a graphical representation is provided in 4.5.

The linked list implementation shown in 57 establishes a very concise representation of a list, and holds within it thoughtful implications in terms of organizing and accessing sequential data. However, it is limited in communicating why this is a canonical example of such a computational entity, or how did one reach this conclusion among other possible entites.

Looking at 57, one can view the different relationships between parts and wholes: the list item composing the list itself, the head pointer being a

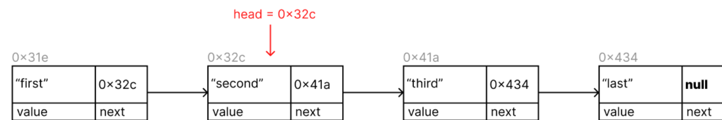

Figure 4.5: The linked list is an abstract data structure which acts as a fundamental conceptual entitiy in computer science. It is here represented as a graph, and implementations can be seen in 57 and 59.

```
struct list_item {
        int value;
        struct list_item *next;
};
typedef struct list_item list_item;

struct list {
        struct list_item *head;
};
typedef struct list list;

size_t size(list *l);
void insert_before(list *l, list_item *before, list_item *item);
```

Listing 57: A textbook example of a fundamental construct in computer science, the linked list. This header file shows all the parts which compose the concept (Kirchner, 2022a).

specific instance of the next pointer, and the different methods to access or modify the list itself. Seeing all of these together suggest an understanding of the whole through the parts which is nowhere explicitly described but everywhere suggested.

Similarly, the example shown in 58 highlights some of the similarities between source code aesthetics and the aesthetics of mathematics. Featured in *Beautiful Code* edited volume, this listing shows the essential components of a regular expression matcher. Regular expressions are a form of linguistic pattern that serve as an input to a regular expression matcher in order to find particular patterns of text in an input string. In this case, the essential components of the matcher are implemented, in a clear and concise way. It highlights the process of looping over an input string, the fundamentals of handling different patterns, and within those the fundamentals of handling different characters in relation to the current pattern. Each part is clearly delineated (and thus fit for its separate purposes) and contributes to an understanding of the whole, by limiting itself to displaying its essence.

Mathematics, like source code, therefore pay close attention to how formal presentation facilitates the cognitive grasping of abstract concepts. Reducing and organizing literal tokens into conceptually coherent units, and meaningful relations to other units—for instance, having the code in 58 reversed, with the `match()` function at the bottom of the document would represent a different level of importance of that entrypoint function, which would complicate the understanding of how the source code functions.

Another overlap between aesthetics in source code and aesthetics in mathematics is the attention that needs to be paid to skill and context. In order to appreciate the regular expression matcher aesthetically, one must know that this is an essential representation of a matcher, and not a functionally complete representation of matcher (58 misses some edge cases to make it usuable). Someone operating in a functional context might find this representation lacking and useless, while concise. Similarly, the

```
/* match: search for regexp anywhere in text */
int match(char *regexp, char *text)
{
    if (regexp[0] == '^')
        return matchhere(regexp + 1, text);
    do
    { /* must look even if string is empty */
        if (matchhere(regexp, text))
            return 1;
    } while (*text++ != '\0');
    return 0;
}

/* matchhere: search for regexp at beginning of text */
int matchhere(char *regexp, char *text)
{
    if (regexp[0] == '\0')
        return 1;
    if (regexp[1] == '*')
        return matchstar(regexp[0], regexp + 2, text);

    if (regexp[0] == '$' && regexp[1] == '\0')
        return *text == '\0';
    if (*text != '\0' && (regexp[0] == '.' || regexp[0] == *text))
        returnmatchhere(regexp + 1, text + 1);
    return 0;
}

/* matchstar: search for c*regexp at beginning of text */
int matchstar(int c, char *regexp, char *text)
{
    do
    { /* a * matches zero or more instances */
        if (matchhere(regexp, text))
            return 1;
    } while (*text != '\0' && (*text++ == c || c == '.'));
    return 0;
}
```

Listing 58: A regular expression matcher by Rob Pike, praised for its elegance and conciseness, but not for its utility (Oram & Wilson, 2007)

```
void remove_cs101(list *l, list_item *target)
{
    list_item *cur = l->head, *prev = NULL;
    while (cur != target)
    {
        prev = cur;
        cur = cur->next;
    }
    if (prev)
        prev->next = cur->next;
    else
        l->head = cur->next;
}

void remove_elegant(list *l, list_item *target)
{
    list_item **p = &l->head;
    while (*p != target)
        p = &(*p)->next;
    *p = target->next;
}
```

Listing 59: A comparison of how to remove an element from a list, with elegance depending on the skill level of the author (Kirchner, 2022b).

linked list example (see 57) might be considered aesthetically pleasing only at a particular level of skill. Indeed, as we see in 59, the distinction is clearly made between a beginner level (labelled "CS101" for the course number of introduction to computer science) and a non-beginner (meaning, elegant) level.

At this point, we should note that some argue for aesthetics as a subset cognitive properties. For instance, Starikova that "*[A]lthough visual representations are involved and understanding does rely on them, it is clearly non-perceptual beauty that initiates aesthetic judgment*" (Starikova, 2018), pointing back to the distinction above as to whether beauty is perceived as intrinsic to the mathematical object, or intrinsic to its representation. Here, we argue that both approaches in source code—intellectual engagement eliciting aesthetic pleasure, and aesthetic pleasure eliciting intellectual engagement—are not mutually exclusive. Specifically, these depend on the nature of background knowledge that the reader holds when

engaging with a program text. On one side, the pre-existence of knowledge allows one to focus on the quality and details of the presentation, such as when mathematicians decide to find more beautiful proofs to an existing theorem. In this case, the knowledge of the theorem, and how its intellectually-perceived simplicity can be translated into a sensually-perceived simplicity and an aesthetic judgment on the form. On the other side, the lack of pre-existing knowledge involves the deciphering of symbols and thus immediate attention to form. Here, the aesthetic judgment precedes the intellectual judgment, all the while not guaranteeing a positive intellectual judgment (e.g. the abstract object, whether program function or mathematical theorem, is presented in an aestheticlly-pleasing manner, but remains shallow, boring, non-sensical or wrong).

We consider here that both intellectual pleasure and aesthetic pleasure happen in a dialogic fashion, considering the symbols and the meanings reciprocally, until intellectual and aesthetic judgments have been given. This is in line with Rota's critique of the term "enlightenment" or "insight" in his phenomenological account of beauty in mathematics. The process of discovery and understanding is a much longer one than a simple stroke of genius experienced by the receiver (Rota, 1997).

An aesthetic experience in mathematics involves uncovering the connections between aesthetic and epistemic value being represented through a mathematical sumbol system. However, such a conception seems to take place as a gradual process of discovery, both from the writer and from the reader. Seen in the light of skill-based aesthetic judgment, this chronological unfolding points towards a final aspect of aesthetics in mathematics specifically, and in the sciences in general: aesthetics as heuristics for knowledge acquisition.

4.4.3 Aesthetics as heuristics

So far, we had been looking at how aesthetics are evaluated in a finished state—that is, once the form of the object (whether a proof or a program text) has stabilized. In doing so, we have left aside a significant aspect of the matter. Aesthetics in mathematics do not need to be seen exclusively as an end, but also as a mean, as a part of the cognitive process engaged to achieve a result. As such, we will see how they also serve as a useful heuristic, both from a personal and social perspectives. Since the ultimate purpose of mathematics specifically, and scientific activities in general, is the establishment of truths, one can only follow that beauty has but a secondary role to play—though that is not to say superfluous.

Complementing the opinions of mathematicians at the turn of the 20tĥ century, Nathalie Sinclair offers a typology of the multiple roles that beauty plays in mathematics. Beyond the one that we have investigated in the previous sections, which she calls the *evaluative* role of beauty, in determining the epistemic value of a conceptual object, she also proposes to look at a *generative* role and at a *motivational* role (Sinclair, 2011). The latter helps the mathematician direct their attention to worthy problems—something which is of limited equivalence in source code, since programming mostly involves external functions. The former holds a guiding role during the inquiry itself, once the domain of inquiry has been chosen. It helps in generating new ideas and insights as one works through a problem. This aesthetic sense can be productive both in its positive evaluations—implying one might be treading a fruitful path—as well as negative—hinting that something might not be conceptually well-formed because it is not formally well-presented[55]. According to Root-Bernstein,

[55]"*The realization that we recognize problems through our anti-aesthetic response to them provides an important clue as to how we go about defining the nature of the problem and recognize its solution. The nature of the disjuncture between our aesthetic expectations and what we observe or think we know reveals the detailed characteristics of the specific problem that presents itself.*" (Root-Bernstein, 2002)

the informal insights of aesthetic intuition precede formal logic. Only when we explicitly recognize that the "tools of thinking" and the "tools of communication" are distinct can we understand the intimate, yet tenuous, connection between thought and language, imagination and logic (Root-Bernstein, 2002).

This is echoed in Norbert Wiener's perception of aesthetics in mathematics as a way to structure a knowledge that is still in the process of being formed, in order to optimize short-term memory as the mental model of the conceptual object being grasped is still being built[56]. This description of a sort of landmark item, in the geographical sense, echoes the role of beacons described by Détienne (Detienne, 2001) and mentioned in 3.2.3, One can therefore consider an aesthetically pleasing element to serve as a sort of beacon used by programmers to construct a mental representation of the program text they are reading or writing.

She positions her argument as a response to the strict focus of the studies in mathematics on the perceptions and reports of highly succesful individuals. If individuals like Poincaré, Hardy and Dirac can self-report their experiences, she inquiries into the ability for individuals of a different skill level to experience generative aesthetics. In a subsequent work, she describes the perception of mathematics students as such:

> *The aesthetic capacity of the student relates to her sensibility in combining information and imagination when making purposeful decisions, regarding meaning and pleasure. (Sinclair, 2011)*

From her investigations, then, it seems that the heuristic value works across skill levels, from Fields medal holder to high-school degree. Doing similar work, Seymour Papert aimed at evaluating the functional role

[56]"*The mathematician's power to operate with temporary emotional symbols and to organize out of them a semipermanent, recallable language. If one is not able to do this, one is likely to find that his ideas evaporate due to the sheer difficulty of preserving them in an as yet unformulated shape.*" quoted by Sinclair in (Sinclair, 2004)

$$\sqrt{2} = p/q \tag{4.1}$$

$$\sqrt{2} * q = p \tag{4.2}$$

$$p = \sqrt{2} * q \tag{4.3}$$

$$(\sqrt{2})^2 = (p/q)^2 \tag{4.4}$$

$$2 = p^2/q^2 \tag{4.5}$$

$$p^2 = 2 * q^2 \tag{4.6}$$

Figure 4.6: Steps of transformation to approach an epistemic value in finding whether or not the square root of 2 is an rational number.

of aesthetics by documenting a group of non-experts working through a proof that the square root of 2 is an irrational number. After a series of transformative steps, the subjects of the study manged to eliminate the square root symbol by elevating the two other variables to the power of two, as in 4.6.

Papert conceptualizes such an observation as a phase of play, a phase of playing which is aesthetic insofar as the person doing mathematics is delimitating an area of exploration, qualitatively trying to fit things together, and seeking patterns that connect or integrate (Papert, 1978), and thus looking to identify parts which would seem to fit a yet-to-be-determined whole. This also seems to confirm the perspective that there are some structures that are meaningul to the mathematician—we present the meaningful structures of source code in 5.2.

An interesting aspect of this conception of aesthetics is their temporal component. While, for evaluative aesthetics, one can grasp the formal representation of the mathematical object in one sweep, this generative role hints at a more prominent temporal component. Both Sinclair and Papert address this shift, and this opens up a new similarity with source code, by shifting from the reader to the writer. One one side, Sinclair connects this unfolding over time with Dewey's theory of inquiry and with Polanyi's

personal knowledge theory, connecting further the psychological perception with the role of aesthetics. Both Dewey and Polanyi offer a conception of knowledge creation which relies particularly on a step-by step development rather than immediate enlightenment (Polanyi & Grene, 1969; Sinclair, 2004); it is precisely this distinction which Papert addresses with his comparison of aesthetics as *gestalt* (evaluative) or *sequential* (generative).

Taking from Dewey's proposal of what an aesthetic experience is[57], we can connect it back to a sequential aesthetic perception in Papert's term, one of learning and discovery, but also to the practice of writing good source code.

In programming practice, the process of working through the establishment of a valuable epistemic object through the sequential change of representations is called *refactoring*. As described by Martin Fowler, author of an eponymous book, refactoring consists in improving the design (while retaining an identical function) of an existing program text. The crux of it is applying a series of small syntactical transformations, each of which help to sharpen the fitness of the parts to which these transformations are applied. Ultimately, the cumulative effect of each of these syntactical transformations ends up being significant in terms of program comprehension, bringing it closer to a sense of elegance (Fowler et al., 1999). We can see a version of this process with a starting point in 2, and a conclusion in 3; all the intermediary steps are described in (Muratori, 2014).

Finally, extending from this personal and psychological perspective on the development of epistemic value through the pursuit of aesthetic perceptions, we can note a last dimension to aesthetics in mathematics: a social component. Shifting our attention away from the modes of mathematical inquiry of individual mathematician, Sinclair and Primm have high-

[57]Dewey presents it as having first and foremost a temporal structure, something that is dynamic, because it takes a certain time to complete, time to overcome obstacles and accumulate sense perceptions and knowledge, following a certain direction, a teleology hopefully concluding in a certain sense of pleasure and fulfillment. (Leddy & Puolakka, 2021)

lighted the practices of the community as a whole, including how truths are named, manipulated and negotiated. (Sinclair, 2011).

On one side, this amounts to uncovering the fact that mathematical problems are being decided upon and researched based on particular values and conventions, conventions which then trickle down into the presentation of results, highlighting trends and social formations both in terms of content of research and style of research (Depaz, 2023). The interpretation provided by Pimm and Sinclair is that aesthetics, through "good taste" subtly reify a power relationship and exclude practitioners by delimiting what is proper writing and proper research (Sinclair & Pimm, 2010). While one could argue for a similar power dynamic when it comes to programming style, one notable difference we see with programming is the highly interactive collaborative environment in which the productive work can be done. Particularly in the case of software engineering, the fact that a given program text is being worked on by different individuals of different skill levels and at different moments suggests a final use of aesthetics as manifested in mathematics, as a social phenomenon. The evaluative posture of the reader in giving a positive value judgment on a given fragment of e.g. a program text or mathematical proof also implies that this positive judgment was given as a generative role (Tomov, 2016). This implies a certain sense of care that was being given to the program text, or to the mathematical proof, which in turn suggests a certain functional quality in the finished object. Beautiful mathematics, as beautiful code, can therefore be seen as a sign that someone cared for others to understand it clearly.

Aesthetics, then, complements more traditional notions of scientific thinking, from representing a mathematical object, enabling access to the conceptual nature and implications of this object, as well as providing useful heuristics in establishing a new object. What remains, and what will be taken up in the next chapter, is to *"reify this meta-logic as a set of rules, axioms, or practices."* (Root-Bernstein, 2002).

In this chapter, we have established a more thorough connection between aesthetics and cognition. First at the philosophical level, we established how source code fits within Nelson Goodman's conception of what is a language of art, before complementing this ability for an aesthetic experience to communicate complex concepts with more contemporary research.

We then moved to more specific domains, examining both how their aesthetic properties engage with cognition, but also how these might relate to those held by source code. Looking at literature, we paid attention to how metaphors, embodied cognition and spatial representations are all devices allowing for the evokation of complex world spaces and cultural references, facilitating the comprehension of (electronic) poetry and prose. Turning to architecture, we went beyond a naïve conception of modernist aesthetics, one which focuses on the plan rather than on the building, to the theories of Christopher Alexander. His concepts of patterns and habitability have been widely transposed in programming practice, highlighting a tension between top-down, abstract design, with bottom-up, hands-on engagement. This notion of direct material engagement led us to further examine how craft folds ties to architecture, and how it facilitates a particular kind of knowledge production—through direct material engagement. Finally, turning to mathematics, we distinguished two main approaches: an evaluative aesthetics, where the representation of a mathematical object has an epistemic function, and a generative aesthetics, which works as a heuristic from a writer's perspective, and often remains unseen to the reader, as it is presented in its final form, without the multiple steps of formal transformations that led to the result.

Throughout, we compared how these specific aesthetic approaches related to source code. Since source code is presented by programmers as existing along these domains of practice, this has allowed us to further refine

a specific aesthetics of source code. The next chapter brings the concepts identified in these domains into a dialogue in order to constitute a coherent view. To do so, we start from source code's material: programming languages.