The role of aesthetics in the understanding(s) of source code

Pierre Depaz

April 2021

1 Introduction

After establishing ground work on aesthetic manifestations in source code for software developers during the Spring 2020 semester, I have concluded with both an empirical manifestation of beautiful code, synthesized a typology of such manifestations—how code can be beautiful—and laid out a preliminary investigation as to why code could be beautiful. Aesthetic manifesations ("beauty") seem to occur whenever facilitate the clarity of intent of the writer, and the agency of the reader, are heightened. Beautiful code makes the underlying concepts clear and easily-graspable, and facilitates its modification by the reader by providing an error-free, and cognitively easy way to do so.

Furthermore, I outlined several directions for further research. These included the exploration of aesthetic standards for two additional categories of code writers: *source code poets* and *hackers*. Following discussions around this outcome, I have added three other directions: literary metaphors, architectural parallels and machine understanding. First, the place of literary metaphors is a response to the cognitive stake at play in reading and writing code, since code can be understood as a formal representation of mental models emerging from complex data structures and

their processing during execution time. Second, the parallels with architecture were suggested by a similar relationship to structure, planning and construction. These parallels are, as we will see below, claimed by software developers themselves, ranging from job titles to commercial best practices of software patterns; on a more theoretical level, the approaches to beauty in architecture will turn out to be productive lenses when thinking not just about executed code, but about source code as well. Third, when claiming that beautiful code facilitates understanding(s), it is important to clarify *whose* understanding of *what*. While previous work has focused on human understandings of human intentions, and human-made concepts, this document investigates to what extent do computers, as concrete machines, understand anything.

I will start by examining instances of source code poetry, defining it, contextualizing it, and analyzing it through close-readings. This will allow us to highlight specific aesthetic standards emerging from this corpus, namely *semantic layering* and *procedural rhetoric*. Source code poetry, with this clear emphasis on *poetry*, will then allow us to address the traditional relationship between literature and code, on an artistic level as well as on a linguistic one. The two concepts mentioned above will lead to an examination of the metaphor, from a literary and from a cognitive standpoint.

Particularly, the relationship that metaphors maintain to the process of knowing and understanding will be highlighted both in human texts and in program texts[1]. Connecting it to mental models will allow us to start thinking of these program texts in terms of *structure*, both surface-structure and deep-structure, and address how a theoretical framework of aesthetics might be connecting the two, including the place of imagination in acquiring knowledge and building understanding in these texts.

Mentioning structure will thus lead us into the overlap between architecture and software. After a short overview of how the two are usually related, I examine a particular set of aesthetic standards developed by Christopher

Alexander in his work on pattern languages. At the cursory level, these are tied to software patterns, techniques for developing better software that have emerged more out of practice rather than out of theory. At a deeper level, we will see that the standards of beauty—or, rather, of this *Quality Without a Name*—can be applied productively to better understand what qualities are exhibited by a program that is deemed beautiful. In particular, Richard P. Gabriel's work will further provide a connection between software, architecture and poetry.

One particular aspect of architecture—the folly, the pavillion, and to some extent large-scale installation artworks—will allow us to transition into our next corpus: hacking. Hacking, defined further as seemingly-exclusively functional code will further requalify the need for aesthetics in source code. We will see how this practice is focused much less on human understanding than on machine understanding, on producing code that is unreadable for the former, and yet crystal-clear for the latter—with an emphasis on human and machine performance. Despite a current lack of extensive research on hacking-related program texts, we will look into two instances of these: the one-liner and the demo to support our investigation in this domain.

This brings us to the broader question of human understanding and machine understandings. Starting from the distinction between syntax and semantics, I highlight discrepancies between semantics in natural languages and semantics in programming languages to define machine understanding as an autotelic one, completely enclosed within a formal description. Coming back to Goodman, we will see how such a formal system fits as a *language of art*, and yet remains ambiguous: is computation exclusively concerned with itself, or can it be said that it relates to the rest of the (non-computable) world? Additionally, the question of aesthetics within programming languages themselves will be approached in a dual approach: as linguistic constructs presenting affordances for creating program texts which exhibit aesthetic properties, and as objects with aesthetic proper-

ties themselves. Whether or not we can agree on machine understanding, the formalism of programming languages, and their aesthetic possibilities, provide an additional perspective on the communication of non-obvious concepts inherent to computing.

In conclusion, we will see that aesthetics in code is not exclusively a literary affair in the strict sense of the term, but is rather at the intersection of literature, architecture and problem-solving, insofar as they manifest through the (re-)presentation of complex concepts and multi-faceted uses, involving their writers and readers in semantics-heavy cognitive processes and mental structures.

Finally, I suggest further directions for research.

---

2 Programming and literary theory

This section focuses on source code poetry, as the closest use of ”literary arts” involving code. We will see how this particular way of writing software, to an explicitly aesthetic end, rather than a functional one, summons specific claims to art and beauty. These claims maintain a complex relationship to the nature and purpose of code, in certain ways embracing the former, and moving away from the latter, but nonetheless allow us to more clearly define such a nature and such purposes. After an overview of the field, including delimitation of our corpus, I will highlight and analyze particular source code poems, chosen for their meaning-making affordances, and conclude on the aesthetic standards at play in their reading and writing, expanding on notions of *double-meaning* and *double-coding*.

2.1 Computer-aided literature

Source code poetry is a distinct subset of electronic literature. A broad field encompassing natural language texts taking full advantage of the dynamic feature of computing to redefine the concept of text, authorship and readership, it nonetheless encompasses a variety of approaches, including generative literature, interactive fiction, visual poetry, source code poetry and esoteric programming languages, as well as certain aspects of software art. However, one of the distinctions that can be made in defining the elements of electronic literature which are included in our corpus is, in line with the framework of this research, the shift from output to input, for executable binary to latent source.

2.1.1 Literature through executed code

A large section of the works which fall within electronic literature focus on the result of an executed program, often effectively obfuscating one of the

many chained acts of writing[1] which allow for the very existence of these works. For instance, the influence of *Colossal Cave Adventure*[2], the first work of interactive fiction, has been centered around on the playable output of the software, rather than on its source code. Written in FORTRAN 4 between 1975 1977, it exhibits several features which wouldn't fit within the typology we've previously established, particularly in terms of variable naming (e.g. variables such as 'KKKT', 'JSPK'; or 'GOTO' statements, whose harm has been considered at the same time this code was written[2]). *Colossal Cave Adventure*'s source code was indeed only examined due to the recognition of the cultural influence of the game, decades later, and not for its intrinsic properties.

A more contemporary example would be that of the Twine game engine, lowering the barrier to entry for writing interactive fictions in the age of the hyperlink. The result, while aesthetically satisfying, widely recognized and appreciated by the interactive fiction community, nonetheless consists in a single HTML document, comprising well-formatted and understandable HTML and CSS markups, along with three single lines of "uglified" JavaScript[3]. The explicit process of uglification[4] relies on the assumption that no one would, or should, read the source code.

In the case of visual poetry, one can see how the source code of works such as bpNichol's *First Screening*[5], is dictated exclusively by the desired output, with a by-product of visually pleasing artifacts throughout the code as foreshadowing the result to come[6]. It is a literal description of static,

[1] See: Béatrice Fraenkel on chains of writing

[2] retrieved from: https://jerz.setonhill.edu/if/crowther/advf4.77-03-11

[3] For instance, the source code of https://pierredepaz.net/-/who/ consists of three lines of 52980 characters, and only 682 whitespace characters

[4] We could expand on this process of uglification, which consists of compacting humanly-laid out source code into the small possible number of characters, usually for a production-ready build, optimized for loading times and dependency processing.

[5] https://www.vispo.com/bp/download/FirstScreeningBybpNichol.txt

[6] Still, a lovely artefact is the subroutine at line 1600, an "offscreen romance" only visible in the source.

desired output, more akin to a cinematic timeline editor, in which there is a 1:1 relationship between the clips laid out and the final reel, and no room for unexpected developments. While computer-powered, such an example of visual poetry tend to side-step the *potentiality* of computing, of which source code is one of the descriptive symbol systems: each execution of the code is going to be exactly the same as the previous one, and the same as the next one[7]. While this might be a drastic example, in which unknowns are reduced to a minimum, visual poetry and interactive do rely heavily on the dynamic aspect of computer procedures to create aesthetic experiences[8]. The difference I am making here is that such aesthetic experience are claimed to take place in the realization of the computer-aided potentials of the work, rather than in the textual description of these potentials[9]. These examples, while far from being exhaustive, nonetheless show how little attention is paid to the source code of these works, since they are clearly—and rightly so—not their most important part.

Computer poetry, an artform based on the playful *détournement* of the computer's constraints, gets closer to our topic insofar as the poems generated represent a more direct application of the rule-based paradigm to the syntactical output of the program. Starting with Christopher Stratchey's love letters, generated (and signed!) by MUC, the Manchester Univac Computer, computer poems are generated by algorithmic processes, and as such rely essentially on this particular feature of programming, laying out rules in order to synthesize syntactically and semantically sound natural language poems. Here, the rules themselves matter as much as the output, a fact highlighted by their ratio: a single rule for a seemingly-infinite amount of outputs.

These works and their authors build on a longer tradition of rule-based

[7] Barring any programmer-independent variables, such as hardware and software platform differences.

[8] For instance, see Text Rain, by Camille Utterbach and Romy Achituv

[9] Tellingly, the Smithsonian Museum, which acquired Text Rain, makes no mention of the source code of the piece.

composition, from Hebrew to the Oulipo and John Cage's indeterministic composition, amongst many others[3], a tradition in which creativity and beauty can emerge from within a strict framework of formal rules. Nonetheless, the source code to these works is rarely released in conjunction with their output, hinting again at their lesser importance in terms of their overall artistic values. If computer poetry is composed of two texts, a natural-language output and a computer-language source, only the former is actually considered to be poetry, often leaving the latter in its shadow (as well as, sometimes, it's programmer, an individual sometimes different from the poet). The poem exists through the code, but isn't exclusively limited to the humanly-readable version of the code, as it only comes to life and can be fully appreciated, under the poet's terms, once interpreted or compiled. While much has been written on computer poetry, few of those commentaries focus on the soundness and the beauty of the source as an essential component of the work, and only in recent times have we seen the emergence of close-readings of the source of some of these works for their own sake[10]. These do constitute a body of work centered around the concept of generative aesthetics[4], in which beauty comes from the unpredictable and somewhat complex interplay of rule-based systems, and whose manifestations encompass not only written works, but games, visual and musical works as well; still, this attention to the result make these works fall on the periphery of our current research.

The aspects of electronic literature examined so far still require computer execution in order to be fully realized as aesthetic experiences. We now turn to these works which still function as works of explicit aesthetic value primarily through the reading of their source. We will examine obfuscated code and code poetry (both at the surface level and at the deep level), to finally delimitate our corpus around the last one.

[10] See the publications in the field of Critical Code studies, Software studies and Platform studies.

2.1.2 Literature through source code

One of the earliest instances of computer source written exclusively to elicit a human emotional reaction, rather than fulfill any immediate, practical function, is perhaps the Apollo 11 Guidance Computer (AGC) code, written in 1969[11] in Assembly. Cultural references and jokes are peppered throughout the text as comments, asserting computer code as a means of expression beyond exclusively technical tasks[12], and independent from a single writer's preferences, since they passed multiple checks and review processes to end up in the final, submitted and executed document.

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

Code comments allow a programmer to write in their mother tongue, rather than in the computer's, enabling more syntactic and semantic flexibility, and thus reveal a burgeoning desire for programmers to express themselves within their medium of choice.

At the turn of the 1980s, following the transition to programming from an annex practice to full-fledged discipline and profession, along with the development of more expressive programming languages (e.g. Pascal in 1970, C in 1972), software development has become a larger field, growing exponentially[13], and fostering practices, communities and development styles and patterns[14]. Source code becomes recognized as a text in its

[11] Hamilton et. al., 1969, retrieved from https://github.com/chrislgarry/Apollo-11

[12] See also: "Crank that wheel", "Burn Baby Burn"

[13] Source: `https://insights.stackoverflow.com/survey/2019#developer-profile-_-years-since-learning-to-code`

[14] From Djikstra's Notes on Structured Programming to Knuth's Literate Programming and

own, which can hold qualities and defects of its own, and to which engineering and artistic attention must be paid. No longer a transitional state from formula to binary, it becomes a semantic material, whose layout, organization and syntax are important to the eyes of its writers and readers. Pushing further into the direction of the visual layout of the code, such an endeavour becomes pursued for its own sake, equally important to the need for a program to be functional.

The Obfuscated C Code Contest[15] is the most popular and oldest organized production of such code, in which programmers submit code that is functional and visually meaningful beyond the exclusive standards of well-formatted code. If the source code's meaning was previously entirely subsumed into the output in computer poetry, and if such a meaning existed in parallel in the comments of the AGC routines, pointing at the overlay of computer-related semantics (e.g. line numbers) and human-related semantics (e.g. number of the beast), obfuscated code is a first foray into closely intertwining these separate meanings in the source code itself, making completely transparent, or completely opaque what the code does just by glancing at it.

Martin's Clean Code

[15] `https://ioccc.org`

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

The above submission to the 1988 IOCCC[16] is a procedure which does exactly what it shows: it deals with a circle. More precisely, it estimates the value of PI by computing its own circumference. While the process is far from being straightforward, relying mainly on bitwise arithmetic operations and a convoluted preprocessor definition, the result is nonetheless very intuitive—the same way that PI is intuitively related to PI. The layout of the code, carefully crafted by introducing whitespace at the necessary locations, doesn't follow any programming practice of indentation, and would probably be useless in any other context, but nonetheless represents another aspect of the *concept* behind the procedure described, not relying on traditional programming syntax[17], but rather on an intuitive, human-specific

[16] Source: `https://web.archive.org/web/20131022114748/http://www0.us.ioccc.org/1988/westley.c`

[17] For such a program, see for instance: `https://crypto.stanford.edu/pbc/notes/`

understanding[18].

Obfuscating practices, beyond their technical necessities (for security and efficiency), are traditionally tied to hacking practices, prominently with one-liners. As such, they rely on the brain-teasing process of deciphering, and on the pleasurable, aesthetic experience of resolving and uniting two parallel meanings: what we see in the code, and what it does[19]. What we focus on here is the aspect of obfuscation which plays with the different layers of meaning: meaning to the computer, meaning to the human, and different ways of representing and communicating this meaning (from uglifying, to consistent formatting, to depicting a circle with dashes and underscores). While the aesthetics at play in hacking will be further explored below, we focus on the fact that obfuscating code practices, beyond hiding the meaning and the intent of the program, also manifest an attempt to represent such a meaning in different ways, leaving aside traditional code-writing practices and suggesting the meaning of the program by challenging the abilities of human interpretation at play in the process of deciphering programs.

2.2 Source code poetry

It is this overlap of meaning which appears as a specific feature of source code poetry. In a broad sense, code poetry conflates classical poetry (as strict syntactical and phonetical form, along with poetic expressivity) with computer code, but it is primarily defined by the fact that it does not require the code to be executed, but only to be read by a human. Following the threads laid out by computer poetry and obfuscated code, code poetry starts from this essential feature of computers to work with strictly defined formal rules, but departs from it in terms of utility. Source code poems are

`pi/code.html`

[18] Concrete poetry also makes such a use of visual cues in traditional literary works.

[19] Also known informally as the "Aha!" moment, crucial in puzzle design.

only functional insofar as they are accepted by the intepreter or compiler of the language in which they are written. To the computer, they are indeed functional, in that they are legal and can be parsed; but they do not do anything of *use*. Such formal compliance is only a pre-requisite, a creative constraint, for their human writers.

Within this reliance on creative constraints provided by a computing environment, the emphasis here is on the act of reading, rather than on the act of deciphering, as we've seen with obfuscated code (and in functional code in general). Source code poems are often easy to read[20], and have an expressive power which operates beyond the common use of programming. Starting from Flusser's approach, I consider poetry as a means to bring concepts into the thinkable, and to crystallize thoughts which are not immediately available to us[5]; through various linguistic techniques, poetry allows us to formulate new concepts and ideas, and to shift perspectives.

In their different manifestations, code poems make the boundary between computer meaning and human meaning thinner and thinner, a feature often afforded by the existence and use of higher-level programming languages. With the development of FLOWMATIC in 1955 by Grace Hopper, it was shown that an English-like syntactical system could be used to communicate concepts for the computer to process. From there, programming languages could be described along a gradient, with binary at the lowest end, and natural language (in an overwheling majority, English) at the highest end. This implies that they could be written and read similarly to English, including word order, pronouncation and interpretation, similar to the error-tolerance of human laguages, which doesn't make the whole communication process fail whenever a specific word, or a word order isn't understood.

[20] See perl haikus in particular

2.2.1 Static code poetry

The community of programmers writing in Perl[21] has been one of the most vibrant and productive communities when it comes to code poetry. Such a use of Perl started in 1990, when the language creator Larry Wall shared some of the poems written in the language, and it gained further exposition through the work of Shannon Hopkins[6]. The poem *Black Perl*, submitted anonymously, is a representative example of the productions of this community:

```
#!/usr/bin perl
no warnings;

BEFOREHAND: close door, each window & exit; wait until time.
  open spellbook, study, read (scan, $elect, tell us);
write it, print the hex while each watches,
  reverse its, length, write, again;
kill spiders, pop them, chop, split, kill them.
  unlink arms, shift, wait & listen (listening, wait),
  sort the flock (then, warn "the goats" & kill "the sheep");
kill them, dump qualms, shift moralities,
  values aside, each one;
die sheep? die to : reverse { the => system
  ( you accept (reject, respect) ) };
next step,
  kill 'the next sacrifice', each sacrifice,
  wait, redo ritual until "all the spirits are pleased";
do { it => "as they say" }.
  do { it => (*everyone***must***participate***in***forbidden**s*e*
       x*)
+ }.
  return last victim; package body;
exit crypt (time, times & "half a time") & close it,
  select (quickly) & warn your (next victim);
AFTERWARDS: tell nobody.
  wait, wait until time;
wait until next year, next decade;
sleep, sleep, die yourself,
  die @last
```

The most obvious feature of this code poem is that it can be read by anyone, including by readers without previous programming experience: each word is valid both as English and as Perl. A second feature is the abundant use of verbs. Perl belongs to a family of programming languages grouped

[21] See: perlmonks, with the spiritual, devoted and communal undertones that such a name implies.

under the *imperative* paradigm, which matches a grammatical mood of natural languages, the *imperative mood*. Such mood emphasizes actions to be take rather than, for instance, descriptions of situations, and thus sets a clear tone for the poem. The fact that Perl is based on stating procedures to be executed and states to be changed creates this feeling of relentless urgency when reading through the poem, a constant need to be taking actions, for things to be changed. Here, the native constraints of the programming language interacts directly with the poetic suggestion of the work in a first way: the nature of Perl is that of giving orders, resulting in a poem which addresses *someone* to execute *something*. Still, Perl's flexibility leaves us wondering as to who and what are concerned by these orders. Is the poem directing its words to itself? To the reader? Is Perl just ever talking exclusively to the computer? This ambiguity of the adressee adds to the ominousness of each verse.

The object of each of these predicates presents a different kind of ambiguity: earlier versions of Perl function in such a way that they ignore unknown tokens[22][23]. Each of the non-reserved keywords in the poem are therefore, to the Perl interpreter, potentially inexistant, allowing for a large latitude of creative freedom from the writer's part. Such a feature allows for a tension between the strict, untoucheable meaning of Perl's reserved keywords, and the almost infinite combination of variable and procedure names and regular expressions. This tension nonetheless happens within a certain rhythm, resulting from the programming syntax: `kill them, dump qualms, shift moralities`, here alternating the computer's lexicon and the poet's, both distinct and nonetheless intertwined to create a *Gestalt*, a whole which is more than the sum of its parts.

A clever use of Perl's handling of undefined variables and execution order allows the writer to use keywords for their human semantics, while sub-

[22] e.g. undefined variables do not cause a core dump.

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

verting their actual computer function. For instance, the `die` function should raise an exception, but wrapped within the `exit ()` and `close` keywords, the command is not interpred and therefore never reaches the execution point, bypassing the abrupt interruption. The subversion here isn't purely semiotic, in the sense of what each individual word means, but rather in how the control flow of the program operates—technical skill is in this case required for artistic skill to be displayed.

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

This instance of Perl poetry highlights a couple of concepts that are particularly present in code poetry. While it has technical knowledge of the language in common with obfuscation, it departs from obfuscated works, which operate through syntax compression, by harnessing the expressive power of semiotic ambiguity, giving new meaning to reserved keywords. Such an ambiguity is furthermore bi-directional: the computing keywords become imbued with natural language significance, bringing the lexicon of the machine into the realm of the poetic, while the human-defined variable and procedure names, and of the regular expressions, are chosen as to appear in line with the rhythm and structure of the language. Such a work highlights the co-existence of human and machine meaning inherent to any program text[24].

[24] Except perhaps those which deal exclusively with scientific and mathematical concepts

2.2.2 Dynamic code poetry

```
class Proc
                           def in_discomfort?; :me; end
                                                                    end
you_are = you =

   ->(you) do
       self.inspect until true
         until nil
               break you
                                                      end
           puts you.in_discomfort?
             you_are[you]
                                                  end

you[
          you_are
]
```

The poem above, written in Ruby by maca[25] in 2011 and titled `self_inspect.rb`, opens up an additional perspective on the relationship between aesthetics and expressivity in source code. Immediately, the layout of the poem is reminiscent both of obfuscated works and of free-verse poetry, such as E.E. Cummings' and Stéphane Mallarmé's works[26]. This particular layout highlights the ultimately arbitrary nature of whitespace use in source code formatting: `self_inspect.rb` breaks away from the implicit rhythm embraced in *Black Perl*, and links to the topics of the poem (introspection and *unheimlichkeit*) by abandoning what are, ultimately, social conventions, and reorganizing the layout to emphasize both keyword and topic, exemplified in the `end` keyword, pushed away at the end of their line.

The poem presents additional features which operate on another level, halfway between the surface and deep structures of the program text. First, the writer makes expressive use of the syntax of Ruby by involving data types. While *Black Perl* remained evasive about the computer semantics of the variables, such semantics take here an integral part. Two data types, the array and the symbol are used not just exclusively as syntactical necessities (since they don't immediately fulfill any essential purpose), but

[25] `https://github.com/maca`

[26] Particularly *Un coup de dés jamais n'abolira le hasard*.

rather as semantic ones. The use of `:me` on line 2 is the only occurence of the first-person pronoun, standing out in a poem littered with references to `you`. Symbols, unlike variable names, stand for variable or method names. While `you` refers to a (hypothetically-)defined value[27], a symbol refers to a variable name, a variable name which is here undefined. Such a reference to a first-person pronoun implies at the same time its ever elusiveness. It is here expressed through this specific syntactic use of this particular data type, while the second-person is referred to through regular variable names, possibly closer to an actual definition. It is a subtlety which doesn't have an immediate equivalent in natural language, and by relying on the concept of reference, hints at an essential *différance* between you and me.

Reinforcing this theme of the elusiveness of the self, maca plays with the ambiguity of the value and type of `you` and `you_are`, until they are revealed to be arrays. Arrays are basic data structures consisting of sequential values, and representing `you` as such suggests the concept of the multiplicity of the self, adding another dimension to the theme of elusiveness. The discomfort of the poem's voice comes from, finally, from this lack of clear definition of who `you` is. Using `you_are` as an index to select an element of an array, subverts the role suggested by the declarative syntax of *you are*. The index, here, doesn't define anything, and yet always refers to something, because of the assigment of its value to what the lambda expression `->` returns. This further complicates the poem's attempt at defining the self, returning the reverse expression `you_are[you]`. While such an expression might have clear, even simple, semantics when read out loud from a natural language perspective, knowledge of the programing language reveals that such a way to assign value contributes significantly to the poem's expressive abilities.

A final feature exhibited by the poem is the execution of the procedure. When running the code, the result is an endless output of print statements

[27] A variable name can represent a value and/or a memory address

of ”me”, since Ruby interprets the last statement of a program as a return value to be printed:

```
...
me
me
me
me
me
me
me
me
me
me
me
me
me
me
me
me
Traceback (most recent call last):
        11913: from poem.rb:16:in ‘<main>’
        11912: from poem.rb:13:in ‘block in <main>’
        11911: from poem.rb:13:in ‘block in <main>’
        11910: from poem.rb:13:in ‘block in <main>’
        11909: from poem.rb:13:in ‘block in <main>’
        11908: from poem.rb:13:in ‘block in <main>’
        11907: from poem.rb:13:in ‘block in <main>’
        11906: from poem.rb:13:in ‘block in <main>’
         ... 11901 levels...
            4: from poem.rb:13:in ‘block in <main>’
            3: from poem.rb:13:in ‘block in <main>’
            2: from poem.rb:12:in ‘block in <main>’
            1: from poem.rb:12:in ‘puts’
self_inspect.rb:12:in ‘puts’: stack level too deep (SystemStackError)
```

The computer execution of the poem provides an additional layer of meaning to our human interpretation. Through the assignment of `you_are` in an `until` loop, the result is an endless succession of the literal interpretation of the symbol `:me`, the actual result of being in discomfort. While we’ve seen that a symbol only refers to something *else*, the concrete[28] output of the poem evokes an insistence of the literal self, exhibiting a different tone than a source in which the presence of the pronoun *you* is clearly dominant. Such a duality of concepts is thus represented in the duality of a concise source and of an extensive output, and is punctuated by the ultimate impossibility of the machine to process the accumulation of these

[28] Both in terms of actual, and in terms of concrete poetry

intertwined references to *me* and *you*, resulting in a stack overflow error.

The added depth of meaning from this code poem goes beyond the syntactic and semantic interplay immediately visible when reading the source, as the execution provides a result whose meaning depends on the coexistence of both source and output. Beyond keywords, variable names and data structures, it is also the procedure itself which gains expressive power: a poem initially about *you* results in a humanly infinite, but hardware-bounded, series of *me*[29].

2.3 Theoretical frameworks of source code poetry

These analyses of program texts have highlighted some of the aesthetic features of source code which can elicit a poetic experience during both reading and execution. These can be further qualified through several concepts, which I introduce and extend here.

The first is **double-meaning**, taken from Camille Paloque-Bergès's work on networked texts, and her analysis of code poetics[7]. She defines it as the affordance provided by the English-like syntax of keywords reserved for programming to act as natural-language signifiers. As we've seen in *Black Perl*, the Perl functions can indeed be interpreted as regular words when the source is read as a human text. Starting from her analysis of *codeworks*, a body of literature centered around a créole language halfway between humanspeak and computerspeak[30], it can be extended into the aesthetically productive overlap of syntactic realms.

Previous research by Philippe Bootz has also highlighted the concept of the *double-text* in the context of computer poetry, a text which exists both in its prototypal, virtual, imagined form, under its source manifestation, and which exists as an instantiated, realized one[8]. However, he asserts

[29] Another productive comparison could be found in Stein's work, *Rose is a rose is a rose...*

[30] See in particular the work of Alan Sondheim and mezangelle

that, in its virtual form, "a work has no reality", specifically because it is not realized. Here again, we encounter the dependence of the source on its realized output, indeed a defining feature of the generative aesthetics of computer poetry. As we've seen in the `self_inspect.rb` poem, a work of code poetry can very much exist in its prototypal form, with its output providing only additional meaning, further qualifying the themes laid out in source beforehand. Indeed, the output of that poem would have a drastically diminished semantic richness if the source isn't also read and understood. For this double-meaning to take place, we can say that the sitation is inverted: the output becomes the virtual, imagined text, while the source is the concrete instantiation of the poem.

Second, we draw on Geoff Cox and Alex McLean's concept of **double-coding**[9]. According to them, double-coding *"exemplifies the material aspects of code both on a functional and an expressive level"* (p.9). Cox and McLean's work, in a thorough exploration of source code as an expressive medium, focus on the political features of speaking through code, as a subversive praxis. They work on the broad social implications of written and spoken[31] code, rather than exclusively on the specific features of what makes source code expressive in the first place. Double-coding nonetheless helps us identify the unique structural features of programming languages which support this expressivity. As we've briefly investigated, notably through the use of data types such as symbols and arrays in source code poetry, programming languages and their syntax hold within them a specific kind of semantics which hold, for those who are familiar with them and understand them, expressive power, once the data type is understood both in its literal sense, and in its metaphorical one. The succint and relevant use of these linguistic features can thicken the meaning of a poem, bringing into the realm of the thinkable ways to approach metaphysical topics.

[31] Which they conflate with the practice of live-coding

Finally, the tight coupling of the source code and the executed result brings up Ian Bogost's concept of **procedural rhetoric**[10]. Bogost presents procedures as a novel means of persuasion, along verbal and visual rhetorics. Working within the realm of videogames, he outlines that the design and execution of processes afford particular stances which, in turn, influence a specific worldview, and therefore arguing for the validity of its existence. Further work has shown that source code examination can already represent these procedures, and hence construct a potential dynamic world from the source[11]. If procedures are expressive, if they can map to particular versions of a world which the player/reader experiences[32], then it can be said that their textual description can also already persuasive, and elicit both rational and emotional reactions due to their depiction of higher-order concepts (e.g. consumption, urbanism, identity, morality). As its prototypal version, source code acts as the pre-requisite for such a rhetoric, and part of its expressive power lies in the procedures it deploys (whether from value assignment, execution jumps or from its overall paradigms[33]). Manifested at the surface level through code, these procedures however run deeper into the conceptual structure of the program text, and such conceptual structures can nonetheless be echoed in the lived experiences of the reader.

We've seen through this section that the poetic expressivity of source code poems rely on several aesthetic mechanisms, which can be combined for further expressive effect. From layout and syntactic obfuscation, to double-meaning through variables and procedure names, double-coding and the integration of data types and functional code into a program text and a rhetoric of procedures in their written form, all of these activate the connection between programming concepts and human concepts to bring the unthinkable within the reach of the thinkable. The next section will explore this connection further, in terms of mental models, literary metaphors

[32] Versions of worlds can be explored further through Goodman's *Ways of Worldmaking*

[33] e.g. declarative, imperative, functional

and cognitive structures.

---

3 Metaphors and Mental Models

This section focuses on metaphors and mental models in both programming and literature, and how they're related to understanding and cognition, both in broad terms and in specific texts. The evokative power of the excerpts seen above make ample use of the multiple facets of understanding, switching from one frame to the other. This switch relates to 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 bypass some of the more minute distinctions 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 they all relate to a larger, more contemporary definition of the concept.

3.1 Theoretical approaches to metaphors

This part of the thesis relies especially on the works of George Lakoff and Mark Johnson, and of Paul Ricoeur, due to their requalification of the nature and role of metaphor in the 20th century. 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, I 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. Following a brief overview of their contributions, I examine the various uses of metaphor in software and in literature, touch upon the cognitive turn in literary studies, and conclude the section by the ambiguity of a cognitive account of programming.

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. Some of these sources are called *schemas*, and 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[12], providing both diversity and reliability. As we will see below, their approach allows us to focus not just on textual objects, but on the vast range of metaphors used specifically in computing-related environments. Given that the source of the metaphor should be grounded, with as little invariablity as possible, in order to qualify a potentially ill-defined target domain, this provides us with a first foray into the inherent elusiveness and instability of computing when presented to a broader audience.

Going beyond the role of metaphors manifested in expressions such as *the desktop*, *the mouse*, or *the cloud*, we will explore Lakoff's understanding of the specifically poetic metaphor further below as preliminary work to assess the linguistic component of computing—source code. For now, to complement his broadening of the metaphorical field, we turn to Paul Ricoeur's assessment of it.

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[13]. 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, one-to-one relationship[34]. Looking at it from the perspective of the sentence brings this rich network of potential meanings and broadens the scope for interpretation. As we've briefly touched upon in the previous section when reading `self_inspect.rb`, all of the evocative meaning of the poem isn't 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)[35]. 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. As we will be approaching the topic of syntax and semantics in programming languages, we will see that these tensions can be a fertile ground for poetic creation through aesthetic manifestations.

[34] As he sees it in the traditional, Aristotelician sense of the term.

[35] For the extent to which source code can be considered discourse has been discussed, see: Cox and McLean, *Speaking Code*.

3.2 Metaphors in literature

3.2.1 Functions of metaphors

If the conceptual turn initiated by Lakoff and Johnson's analysis of the metaphor broadens the horizon of their applicability beyond the strict domain of literature, it is nonetheless clear that metaphors appear and operate in particular ways in literary works, from fiction to poetry. We look at such specificity here in anticipation of identifying which features of poetic metaphors could be mapped to the program texts of our corpus—whether explicitly poetic, as in source code poetry, or not, as in regular source code.

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. This echoes our mention of Flusser's analysis of poetry as that which brings ideas into the realm of the thinkable.

It does so 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 associated with brightness by saying *"Juliet is the sun*[36].

Poetic metaphors can also *elaborate*, by adding more dimensions to the target domain, while nonetheless being related to its original dimen-

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

sion. 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 Camus describes Tipasa's countryside as being *blackened from the sun*[37], it subverts our pre-conceptions about what the countryside is, what the sun does, and hints at a semantic depth which would go on to support a whole philosophical thought (*la pensée de midi*). 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.

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 metaphor of Charles Bovary's cap, a drawn-out metaphorin Flaubert's work which ends up depicting something which clearly isn't a cap, operates by extending the literal understanding of how

[37] *"A certaines heures, la campagne est noire de soleil"*, from *Noces à Tipasa*

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. 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[38], or for the novel itself, whose structure itself is composed of complex layers, under the guise of banal appearances. Composed metaphors highlight how they exist along *degrees of meanings*, from the conventional to the poetic, and further to the non-sensical. This difference of degree, rather than of kind, is one I ascribe to when it comes to delimiting corpus of the present research in different ways of writing and reading code—writing code as poetry, as tool, as a hack or as research aren't absolutely siloed off from each other.

Through these, Lakoff and Johnson highlight how metaphors *function*, and how they 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 are ultimately seen as a means to comprehend the world. By importing structure from the source, the metaphor in turn creates structure in our lives, in our understandings (and thus have power over us). Our understanding grasps these structures through their features and attributes (one might even call them affordances, following Gibson[14]), and integrates them as a given—in what Ricoeur would call a *dead* metaphor. This is one of their key contribution, that 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.

This approach of beauty as means to understand however predates

[38]Which ultimately leads Emma to her demise.

Lakoff and Johnson. 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, and taken later on by Ricoeur as a basis for his study, *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. Ricoeur's theory of the metaphor indeed builds on Beardsley's conception that metaphor can have a designative role (the primary subject) which adds a *"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. For Ricoeur, it is indeed literary criticism, beyond logical grammar and linguistics, which hold the key to understanding metaphors. Through an analysis of Beardsley's work, he highlights the 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"[39]

As such, Beardsley highlights the possibility of an aesthetic experience to make understandable, to unlock new knowledge in the beholder, and

[39] The Aesthetic Experience, in The Aesthetic Point of View[15].

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 metaphor creating 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.

3.2.2 Literature and cognitive structures

More recent work in aesthetics and literary research have continued in this direction. Building on the focus on conceptual structures, the attention 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 explaing "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?*. 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, and thus a good starting point for a new contribution to understanding the arts. A similar approach, related to scientific knowledge, can be seen in Michael Polanyi's work on tacit knowledge, in which that which the scientist knows isn't entirely 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)[16].

Some of the cognitive benefits of art aren't too dis-similar to those posed by Beardsley, but shift their rationale from strict hermeneutics and criticism to cognitive science. These benefits can be pleasure, emotion, or understanding. Terrence 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.

Harris 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 bagage (a personal encyclopedia) brought by the reader. Such a bagage consists of varying experience levels, of quality of the know-how that is to be activated during 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, imagina-

tion 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*[40] of literary works, which she defines as (linguistic) simulations relying on the senses to communicate situations, concepts, and potential realities. 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[41]. It seems that the imaginative process depends in part on visual and spatial projections, and suggests the 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 code has close to no 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 abstraction

[40] Elane Scarry's expression

[41] 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 really loud, over and over again", source: `https://medium.com/artists-and-machine-intelligence/a-journey-through-multiple-dimensions-and-transformations-in-space-the-final-frontier-d8435d81ca51`

processes, including metaphor.

3.3 Metaphors in Software

3.3.1 User-facing metaphors

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 computers (laptops, tablets, smartphones, etc.), and thus definitely within the realm of machines, the Turing machine isn't strictly a machine *per se*. Rather, it is more accurately defined as a mathematical model which in turn defines an abstract machine. 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[42]. Debates in computer science related to the nature of computing[17] 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.

Jumping ahead to the 1980s, 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 including icons and folders. For instance, the desktop metaphor relies on previous understand-

[42] See research in biological computing, using DNA and protein to perform computational tasks

ing 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 (such as the Star), 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 Sony Walkman, makes a similar statement when he describes the Sony Walkman, a never-before-seen compound of technological innovations, in terms of pre-existing, and well-established technologies[18]. 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 sense of the broad possibilities brought forth by the computing revolution.

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 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 role of the metaphor. These depictions of things-as-other-things influence the mental model which we build of the computer system we engage in. 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 when he bases it on Jameson's theory of cognitive mapping. While Jameson uses it in a political and historical context, the heuristic is nonetheless useful here: 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. Substituting ideology with the computer[43], we can see how such a process helps make sense of the unthinkable, of that which is too complex to grasp and therefore must be put into symbols (words, icons, sounds, etc.).

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 that of abstraction. 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.

This need to get away from the specificities of the machines has been one of the essential drives in the development of programming languages. Since we cannot easily and intuitively deal with binary notation to represent complex concepts, programming helps us deal with this hurdle by presenting things in terms of other things. Most fundamentally, we represent binary signs in terms of English language (e.g. from binary to Assembly). 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

[43] 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*.

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 (i.e. conceptual) 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, I argue that metaphors have played a part in helping programmers construct useful mental representations related to computing. These metaphors can go both ways: helping humans understand computing concepts, and to a certain extent, helping computers understand human concepts.

3.3.2 Programmer-facing metaphors

Perhaps one of the first metaphors a programmer encounters when learning about the discipline is that which states that the 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). The difficulty in

explaining, in that context, the need for a *void* keyword to individuals with limited experience and knowledge of how programming works is a good example of the non-straightforwardness of computing concepts. 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. Incidentally, a server is also a style of software architecture, to which we will return later.

Another instance of synbolic 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, 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 hint at the need to represent complex concepts into humanly-graspable terms, what Paul Fishwick calls *text-based aesthetics*[19]. 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. The 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[44].

The reverse process also brings forth issues of conceptual representation through formal symbolic means. The work of early artifical intelligence researchers consists 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 (intepretation and generation)[20]. 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. 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.

As we've seen, 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, and therefore too loose for programming languages, heavily based on the analytic tradition. Yet, they still rely heavily on models in order to make human concepts

[44] See the famous comment in the UNIX source: *You are not expected to understand this*.

graspable and operation to the computer. These tools deployed during the representational process differ from conventional or poetic metaphors insofar as they can be logically operated upon and therefore empirically verifiable or falsifiable. These models are means through which we aim at taking the conceptual structures on which metaphors also operate, and explicit them in formal symbol systems[45].

Abstraction, metaphors and synmbolic representations are thus useful tools when it comes to computing, 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).

3.4 Programming and psychology

So metaphors work in software because they do not exist just within literature, and yet remain too vague for a strict computer interpretation. Such a computer interpretation, in turn, is too complex and fine-grained for most individuals interacting with them (from end-users to most programmers and computer scientists) to be useful. The conclusion we establish here, is about how connections between mental models relate to the process of understanding, at the overlap between human understanding and computer understanding, and how aesthetic experience can affect this encounter.

The mental model offers a good starting point for exploring this overlap. A mental model, as a kind of internal symbolic representation of external reality, is a more rigorous and formal conceptual structure than a metaphor—which only offers a broad direction through evokative power, rather than an actionable basis. They are related to knowledge, since the construction

[45]For a further inquiry of models and theories, see Weizenbaum in *Computer Power and Human Reason*

of accurate and useful mental models through the process of understanding underpins knowledge acquisition. However, mental models need not be correlated with empirical truth, but extensive enough to be described by logical means. Mental models can be informed, constructed or further qualified by the use of metaphors, but they are nonetheless more precise than the cognitive structures on which metaphors rely—a mental model can be seen as a more specific instance of a conceptual structure. The term *schema*, used in cognition-influenced literary studies, again following Lakoff, is here a point of entry into the psychology of computer programming.

Francoise Détienne, in her study of how computer programmers design and understand programs[1], defines the activity of designing 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). This flexibility is useful when one needs to deal with two aspects of working within a programming environment. An added complexity to the task of programming comes with 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), one of the tensions at the heart of computer programming. In order to be appropriately dealt with, then, programmers must activate and refine mental models of a program which resolves this tension.

In programming, within a given context—which include 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[21], she nonetheless highlights some differences. In program texts, 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. 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 presctiptive/-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.

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 throuh its *surface structure*[22]. 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. However, one of the questions that isn't answered specifically, and which is the aim of this thesis, is to highlight how does the specific surface structure in programming result in the understanding of the deep structure.

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)[23]. They conclude that, while language processing might not be one of the essential ways that we process code, it also does not rely on exclusively mathematical functions. Stimulating in particular the 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 nor linguistics should be the only lens through which we look at code.

Going back to research in contemporary literary studies can start laying

out threads of an answer. Jérôme Pelletier uses Carl Plantinga to define emotional responses in the face of aesthetic objects as dual: either one has an emotional response to the artefact itself (surface), of an emotional response to what it represents (deep). In the context of reading fiction, the reader is helped in their understanding by looking out for *guides* or *props*[46], which are similar to the *beacons* emphasized by Détienne. A notable difference is that the guides are suggested, implied, left as traces for the reader to subtly construct (as in the case of the cap metaphor in *Madame Bovary*), rather than explicitly stated throughout the program text (usually most obviously in the form of comments). However, we've seen previously that the use of comments is, by most programmers, not considered to be an aesthetic feature of an inspected source code, hinting at the fact that (useful) subtlety, might be a desired attribute of beautiful code.

Programming is then 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. Furthermore, both programming and fiction suggest surface-level guiding points helping the process of constructing mental models and conceptual representations. It is also non-fiction, in that it deals with concrete issues and problems (more often than not, a pestering bug), and that it provides a pragmatic frame for processing representations, in which assumptions stemming from burgeoning mental models can be easily verified or falsified. It might then be appropriate to treat it as such, simultaneously fiction and non-fiction. Finally, 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."*[24].

To conclude this section, then, we turn to Jerome Bruner, who considers that art allows us to *"reading in others' minds"*, to anticipate what a writer has been intending for us to understand through their text, eithe

[46] Currie, 1990

program or narrative. This intent component relates to the interpretation issue mentioned above: the interpretation of the machine is different from the interpretation of the human, and therefore what also needs to be intepreted is the intent of the author. Reading is then akin to constructing a *cognitive cartography*, allowing for an experience to be made intelligible, sensible. The repeated implication of spatial and visual components of metaphors and mental models allows us to consider metaphors as an architecture of thought[25]. The next section is therefore dedicated to examining more closely the parallels between software architecture and physical architecture, and how the aesthetic standards of the latter could apply to the aesthetic standards of the former.

---

4 Architecture and software quality

Beyond its physical manifestation, the term architecture has also been applied to software development. While software does not deal with anything of concrete, immediate existence beyond lines of code (in contrast with hardware), it nonetheless holds similarities with physical architecture insofar as it is about the design, planning and construction of complex structures for human use, and possibly enjoyment. This section takes a closer look at this relationship, and particularly to the place of *patterns*. Patterns, as we will see, can elicit *goodness* in a construction and, through the concrete manifestation of *habitability*, itself of form of beauty.

4.1 General software architecture

Software architecture emerged as a consequence of the structured revolution[26], which was concerned more with the higher-level organization of code in order to ensure the quality of the software produced. Such an assurance was suggested by Djikstra in two ways: by ensuring the provability of programs in a rigorously mathematic approach, and by ensuring that programs remained as readable as possible for the programmers. Structure has therefore been an essential component of the intelligibility of the software since the 1970s. It's only in the late 1990s that software architecture as a discipline has been recognized as such, stemming form a bottom-up approach of recognizing that some ways in which code is organized is better than others.

4.1.1 Top-down software architecture

Today,

> software architectural models are intended to describe the structure and behavior of a system in terms of computational

> entities, their interactions and its composition patterns, so to reason about systems at more abstract level, disregarding implementation details.[27]

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. Jameson indeed borrows the phrase from Kevin Lynch, whose work on *The Image of the City* 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. Garland notes that, in its most effective cases, software architecture can expose the high-level constraints on the design of a system, as well as *the rationale for making specific architectural choices*. Again, the intent of the architect (or the programmer) matters along with a purely descriptive depiction of the system. Intent (along with reuse, construction, evolution, analysis and management) is one of the crucial aspects of the computational paradigm on which the software is built.

As an example, the Linux Kernel's architecture can be considered, amongst others reasons, one of the reasons why the project became so popular once integrated into the GNU ecosystem. Along with its distribution license, two of its defining features are speed and portability. While speed can be attributed to its use of C code, also responsible to some extent for its portability, the architecture of the kernel is separated in multiple components which make its extension *relatively* simple. On one side is the monolithic architecture of the kernel, in which process and memory management, virtual file systems, input/output schedulers, device drivers and network interfaces are all lumped together in kernel space. However, this architecture also allows ofr dynamically loadable kernel modules, pieces of the operating system which can be added and removed to the operating

system without interference with the core features. This provides a quality of extendability which further contributes to the success of the ecosystem of the Linux ecosystem (see the numerous Linux-based distributions, from Ubuntu to Red Hat and Android).

An architecture, such as that of the Linux kernel, thus provides significant *semantic* content about the kinds of properties that developers should be concerned about and the expected paths of evolution of the overall system, as well as its subparts. Other architectures include, for instance, the client-server architecture (with the peer-to-peer architecture as an alternative), the model-view-controller architecture (and its presentation-abstraction-control counterpart), and one can even find their source in chip design, with Friedrich Kittler famously claiming that the last people who ever truly wrote anything where the Intel engineers laying out the plan of the 8086 chip (which would engender the whole family of x86-based devices)[28]. In this case, this instance is one of the few which relates software architecture to its physical counterpart, albeit in a very technical sense of plans and diagrams.

In the literature consulted for this research, there are only few explicit references to beauty in software architecture design. instead, desirable properties are those of performance, security, availability, functionality, usability, modifiability, portability, reusability, integrability and testability. Perhaps this is due to the fact that the understanding of beauty in terms of external manifestation—decoration—isn't here the main point of the endeavour. Following Adolf Loos, and inspecting software architecture diagram and specifications, it seems that in this specific case, ornament seems to be a crime: highlighting the rise of the useful object in modern civilization[47], he argues that there should be no decoration needed to contribute to the functionality of the object, for fear that it must go out of style.

[47] Along with its white-supremacist undertones rampant in Europe at the time

Style is nonetheless present in software architecture. In this context, an architectural style typically specifies a design vocabulary, constraints on how that vocabulary is used, and semantic assumptions about that vocabulary. For example, a pipe-and-filter style might specify a vocabulary in which the processing components are data transformers (filters), and the interactions are via order-preserving streams (pipes). When it comes down to programming such an architectural style, pipes and filters do have a very real existence in the lines of source code. These concepts are inscribed as the `|` character for pipes, or the `.filter()` method on the JavaScript array type, which itself has different ways of being written (e.g. with an anonymous callback function, or an externally defined function). By virtue of there being different ways being written, one can always argue for whether or not one is better than the other, ultimately resulting in better, clearer—and perhaps therefore beautiful—source code.

More specifically, the aesthetic manifestations in the form of source code enter in a dialogue with software architecture. If a good system architecture should first and foremost exhbit conceptual integrity[29], one can extend this integrity to its source code manifestation. A message-passing architecture with a series of global variables at the top of each file, or an HTTP server which also subscribes to event channels, would look ugly to most, since they betray their original concept. These concrete manifestations of a *local texture of incoherence*, to paraphrase Beardsley, might be more akin to a *code smell*, a hint that something in the program might be deeply wrong.

Different kinds of architecture all deal with structures, and these structure can only fit together in a satisfactory way if their componnents relate to each other in a satisfactory way. One could also make the parallel between musical orchestration, and computing orchestration as the automated configuration, coordination, deployment and maintenance of (mostly distributed) computer systems and software. If an orchestra has an architecture, it nonetheless also features virtuosos and aesthetically-

pleasing phrases (dealing, once again, with tension and its resolution); if a novel has an architecture, it also has specific aesthetic manifestations in its sentences; if computer systems have an architecture, they could possibly have a beautiful manifestation in its individual components which, perhaps not originating from its architecture, nonetheless relate closely to it.

4.1.2 Craftsmanship as bottom-up software construction

Speaking of individual components, we can recall that before the architect came the craftsman. 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, in which the craftsman is relegated to the role of executioner, until the arrival of civil engineering and blueprints overwhelmingly formalized the discipline. The classical architect, here, serves as the counterpart to the computer scientist, except in an inverse relation: the architect emerged from centuries of hands-on work, while the computer scientist (formerly known as mathematician) was first to a whole field of practicioners 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 results (e.g. Notre Dame de Paris, Basilica of Sienna). On the other side, letting go of structured and restricted modes of working characterizing computer programming 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[30].

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[31], both putting pride in the quality and beauty of their work.

The relation of craftsmanship to architecture is as important as the relationship of craftsmanship to programming. A traditional perspective on the matter is that of the motor skills, with dexterity, care and experience as essential features of a craftsman's ability to realize something beautiful[32], along with self-assigned standards of quality[33]. These qualitative standards which, when pushed to their extreme, result in a craftsperson's *style*, are to be gained through practice and experience, rather than by explicit measurements[34] [48]. Two things are concerned here: tools and materials[34]. 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).

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.[32]*

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. 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 craftsman-

[48] 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."*

ship, as well as other forms of cratsmanship-based work in an information economy. Beautiful code, code well-written, is indeed an integral part of software craftsmanship[35]. More than just function for itself, code among programmers can, and should be held to beauty standards[36]. Such standards are another relationship with traditional craftsmanship—in this case, form is indeed mostly following function.

A craftsman's material consciousness is recognized by the anthropomorphic qualities ascribed by the craftsman to the material. In the case of code, adjectives such as "clean", "elegant", "smelly" occur over and over in online discussions of programmers. 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. As software craftsmen assemble in loose hierarchies to construct software, the aesthetic standard is *the respect of others*[37].

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[38]. In that sense, code is a material which can be almost seamlessly converted from information to information-*processing*, and vice-versa. Disregarding for now the very real impact of computing on the environment, code as a material is perhaps the only non-finite material that craftspeople can work with—along with words.

Code, then, 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, cre-*

ating 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.[31]

This bottom-up approach to programming, of the craftsman carefully writing and assembling their code into a satsfying artefact, can be examplified in the emergence of *patterns* in software design. Before we examine these more closely, we first turn to a the particular aspect of architectural theory which influenced it, that of Christopher Alexander's.

4.2 Beauty in architecture

4.2.1 A functionalist approach

If it is still unclear whether or not beauty is part of software architecture, beauty is definitely one of the essential components, and aims, of the architect, dating back to Vitruvius's maxim that a building should exhibit *firmitas, utilitas, venustas* (solidity, usefulness, beauty). While structure is meant to, by definition, stand the test of time[49], utility can be assessed by the extent to which a building fulfills its intended function, and beauty remains elusive, more often igniting debates amongst architects, rather than creating consensus over particularly beautiful constructions[50]. Different schools of beauty in architecture exist, which would could very roughly separate between *top-down* or *bottom-up* approaches.

In terms of top-down approaches, we've seen that, since beautiful software is, first and foremost, software which *runs as intended*, perhaps the first architectural standard which we can apply here is that of Louis Sullivan's *form follows function*. Due to its physical manifestation, Sullivan's

[49] See the still standing structures of Roman and Greek antiquities, based on a particular mixture of cement.

[50] With the exception, perhaps of Frank Lloyd Wright's Fallingwater, and Ludwig Mies van der Rohe's Neue Nationalgallerie.

statement is therefore inevitably translated 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.[39]

We must however keep in mind that Sullivan, within the Chicago School of Architecture, was one of the pioneers of the modern skyscraper, and therefore focused on a very particular kind of architecture, in which productivity became the only, explicit function and the context for his statement. 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 by Loos 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 y thing able to spiritualize industrial work and common life[40].

Rancière offers us an additional perspective, departing form the strict function of an object or of a building, to its *use*. Such a shift moves 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."*[41]. The bottom-up approach is therefore one which

might relate less to software architecture as a top-down, abstraction process, and more to an immediate, usable one of the craftsman as a creator who is highly conscious of the user. The work of Lynch, which we've mentioned above, stands in the tradition of various urban planners and architects such as Jane Jacobs and William H. Whythe. Whythe's work, for instance, focused on empirical observation in order to determine what makes a good space, deducing its aesthetic properties (such as flowing water, movable chairs, the presence of street food vendors, multiple-layered sittings, etc.).

4.2.2 A spiritual approach

Christopher Alexander also belongs to this empirical tradition of examining what makes a space *good* or not, by exmaining its uses and the feelings it elicits in the people who tread its grounds. First developing his theory when working on the design of the Bay Area Rapid Transit System, he elaborated an approach of architecture which does not exclusively rely on abstract design, 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[42] [...]

In this work, he focuses on how beauty is involved in moving from *disorganized complexity* to *organized complexity*, an organizing 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, understood as the sensual, formal properties of an object, but rather with the existence of such objects.

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. Architecture as a whole does clearly take into account the role of tension, of which it is yet another manifestation, akin to those we've seen in, amongst others, Ricoeur's analysis of the metaphor, and the resolution of the riddles presented in works of obfuscated source code.

His approach to this quality is successively named as *appropriateness*, *rightness to fit*, *not-simplicity* and *wholeness*. All of these have in common the subsequent need for a purpose, a purpose which he calls the *Quality Without a Name*. This quality, he says, is complicated to name, 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.

Among the adjectives he uses to circle around this quality are *whole, comfortable, free, exact, egoless, eternal*. Since his work applies more broadly to any design-connected discipline, it also applies to software development. Using the word program as an umbrella for *code block*, we can briefly sketch out how such properties could apply to software.

A *whole* program is a program which isn't missing any features, whose encounter (or lack thereof) might cause a crash.

I see two interpretations to a program being *comfortable*. First, it is a program which does not work against its material, a piece of code which is not trying to "re-invent the wheel", when the wheel might already be built-in, or which does indeed re-invent it, if the existing wheels aren't satisfying. Second, a comfortable program is one which might be modified without fear of some unintended side-effects, without inivisible dependencies which might then compromise the whole.

Free programs are programs which, anthropomorphizing set aside for an

instant, lead their own lives, while being mindful of the lives of others in the shared environment (the design philosophy of the UNIX operating system of *"doing one thing well"*, with its consequence of being able to compose these programs into elaborate chains of data processing, can be a good example of such freedom).

Exact programs are, then, programs which do not exhibit any verbosity, in which every line is necessary and required, without being so obscure that it hinders comfort.

A program that is too exact might be exhibiting too much of its writer's *ego*, too specific and requiring a unique kind of background knowledge which other readers might not have[51].

Finally, an *eternal* program relates to the timelessness mentioned in the title of its work—it touches upon the idea of the sublime, a deep, ambivalent feeling of something that stands beyond past, present and future. Programming might be too young of a discipline to be able to single out a precise example, but the Lisp interpreter might be a good candidate, since it is a concise, succint way of writing an interpreter of Lisp in Lisp, embodying the essence of programming language research and some of the main principles of computation (recursion, symbols, interchangeability between data and procedures).

Alexander did conduct 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*[43]. In it, he lists out 253 patterns which, he claims, form a language, akin to a Chomskian generative grammar, reusable and extendable in a very concrete way. This study has had a significant impact on the computer science community, to which we turn to next.

[51] See *egoless programming*, mentionned above.

4.3 Patterns in software

4.3.1 Design patterns

A Pattern Language kickstarted a whole field of research based around this idea 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 and shared with others.

Besides the theoretical similarities 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, then with C++, [52]. The similarity between a pattern and an object, and their promise of using them which would lead to better results on multiple dimensions, made it very attractive to software developers. Writing in *Patterns of Software* (with a foreword by Alexander), Richard P. Gabriel illustrates that point:

> 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

[52] today most of the programming languages allow for some object-oriented paradigm

needs of the computer and the compiler writer.[44]

The real issue raised here in programming seems to be, again, not to speak to the machine, but to speak to other humans. This complexity of communication, had always asked to be solved, perhaps at this point in the form of object-orientation. Understanding software is hard. Creating, identifying, and formalizing patterns into re-usable solutions turns out to be at least as hard[45]. 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.

4.3.2 Compression and habitability through patterns

Throughout his work, Gabriel weaves 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 pattern languages. Two concepts in particular are worth examining a bit further: *compression* and *habitability*.

Compression, in narrative and poetic text, is the process through which a word is given additional meaning through the rest of the sentence. In a sentence such as *"Last night I dreamt I went to Manderley again."*[53], 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

[53] From Daphne DuMaurier, *Rebecca*.

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 if 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 verbosity, is a thin line of being self-explanatory without being verbose.

This recalls the idea of *semantic proximity*, extracted from our analysis of programmers' comments and opinions on what they found makes code beautiful. Such a pattern does however contrast with the nature of object-oriented programming, in which inheritance (and subsequent local abstraction of subclasses) is considered best practice. 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.[44][54]

Finally, Gabriel, writing in 1998, mentions that compression isn't so much a problem in poetry since, ultimately, the definitions of each words aren't quite limited to the poet's own mind but, as we've seen, also existing in the broad conceptual structures which readers hold. However, since all aspects of a program is always by definition explicitly defined, programmers thus have the ultimate say on the definition of most of the data and

[54] He adds that this isn't so much an issue if one is using a powerful and efficient IDE—a remark which opens up the question of the role of tools and technical mediators in the reading and writing process...

functions described in code. Compression doesn't work as well because the reader cannot assume anything that is being mentioned in the code (and defined elsewhere), without risking the (error-raising) consequence of being wrong.

His particular assumption that others will want to modify and extend source code is one that is influenced by his background as a commercial developer. Other pieces of code might just be satisfying in being read or deciphered (as we've seen in source code poetry) but this assumption of interaction with the code brings in another concept, that of *habitability*. In his 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.[44]

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[55]. It relates to Alexander's property of *comfort*, by affording involvement instead of estrangement. A specific instance of habitability, in software patterns, might be difficult to pinpoint, but can pop up in some cases: 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.

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

[55] Recall the developer who mentioned that beautiful code he saw was code which clearly separated hardware dependent sections from hardware independent sections.

> of a building cannot be made alive when they are drawn at a drawing board.[42]

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[46]. 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, they join 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*[56]. If patterns express relations between contexts, problems and solutions, then it seems that one of the main complaints of developers looking at their code and seeing chunks of foreign code dumped in the middle to fix some generic problem[57], is the 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.

The other criticism is that software patterns are often workarounds for features that a particular programming language doesn't allow from the get-go, or offer more convoluted implementations when written in Smalltalk and C++ than, for instance, Lisp[58]. One aspect that has been eluded so

[56] Addressing this concern, the failure of strict top-down hierarchies in software development resulted in the agile methodology for business teams

[57] The example of the best pattern to retro-fit an air conditionner on a building would be a non-problem if the air-conditionning had been designed in from the get-go `https://wiki.c2.com/?PatternsAreNotTheLesserOfTwoEvils`.

[58] Peter Norvig highlights that most patterns in the original book have much simpler imple-

far, and perhaps the most inescapable context of all, is the programming language used. One doesn't write Ruby like one writes Java(tm), or C++, and certainly not Lisp.

To conclude this section, we've 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[59], points not just to an explicit conditioning of form to its function (in which case we would all write handmade Assembly code), but rather to an elusive, yet present quality, which is both problem- and context-dependent. It is 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 aren't, however, explicitly praised for their beauty, perhaps because they disregard these contexts—by definition, they're high-level abstractions. Generic solutions are rarely elegant solutions. Circling back to our investigation of software as craftsmanship, we now turn to examine more closely both the tools and the material of programmers: programming languages.

---

mentations in Lisp, see: urlhttp://www.norvig.com/design-patterns/design-patterns.pdf

[59] even spawning short-lived debates about his quality without a name on stackoverflow: `https://stackoverflow.com/questions/458242/quality-without-a-name-qwan-examples`.

5 Programming languages as semantic material

Programming languages have so far been set aside when examining which sensual aspects of source code resulted in what could be deemed a "beautiful" program text. Since we're focusing on semantics (deep-structure) represented through syntax (surface-structure), and since programming languages are in essence the frame for defining legal syntax, this section examines the influence of programming languages in the aesthetic features of source code. To do, we first go over a broad description of programming languages, concluding on what makes a programming language expressive. Second, we relate their formal aspect to Goodman's *Languages of Art*, and assess whether or not they are a good fit as an artistic, expressive system. Third, we touch upon the problem of semantics in programming languages, and how they might differ from a human understanding of semantics. Finally, we highlight a couple of computing-specific concepts which are made explicit by programming language research, and further define the kinds of concepts that are defined and manipulated when writing code.

All in all, this will allow us to highlight how programming languages engage with the problem of aesthetics and understanding from a systemic point of view, and how they act as an interface between human and machine understanding.

5.1 Theoretical programming languages

A programming language is a strictly-defined set of syntactic rules and symbols for describing instructions to be executed by the processor. The history of programming languages is, amongst others, the history of decoupling the means of creating softwarefrom hardware. The earliest programming languages were embedded in hardware itself, such as piano rolls

and punched cards for Jacquard looms. Similarly, the first electric computers then required manual re-wiring of the mainframes in order to implement any change in the algorithm being computed, which then gave way to the stack of cards fed into the machine (similar to those used to programmed the Apollo landing unit whose source code we've seen above). It is with the shift to the stored-program model, at the dawn of the 1950s, that the programs could be written, stored, recalled and executed in their electro(-mecha)nical form, essentially freeing the software result from any immediately material representation.

This engineering tendency to separate software from hardware saw a parallel in the development of programming languages themselves. Based on Turing's design, any instruction processed by the machine, needs to, ultimately, execute one of the built-in (literally, hardwired) instructions of the processor. Also called *machine language*, these instructions set describe the specific implementation of the most common operations executed by a computer (e.g. `add, move, read, load`, etc.). While these are represented as binary numbers to the processing unit (as is everything else), some of the first programming languages did not require the writer to write those numbers themselves. Instead, they could use a family of languages called Assembly, which are instructions whose syntax is loosely based on English and translates in turn to machine instructions. Considered today as some of the most low-level code one can write, Assembly languages were machine-dependent, featuring a one-to-one translation from English keywords to the kind of instruction sets they were programmed to generate. As such, a program written on a particular model of a computer could not be executed without any modifications on a another machine.

The first widely acknoweldged high-level language which allowed for a complete decoupling of hardware and software was FORTRAN[60]. At this point, programmers did not need to care about the specifics of the machine

[60]With Plankalkül, Short Code and Autocode as partial proposals before it.

that they were running on anymore. Moving away from ”hand-crafted” Assembly code, FORTRAN, and the subsquent COBOL, Lisp and ALGOL 58 also started being concerned with the specific definition of their syntax in a non-ambiguous manner. Using BNF notation, it became possible to formalize their syntactic rules in order to prevent any unxepected behaviour and support rigorous reasoning for the implementation and research of current and subsequent languages. With such specifications, and with the decoupling from hardware, programming languages became, on paper, *context-free*.

The context-free grammatical basis for programming allowed for the further development of compilers and interpreters. These two are themselves binary programs which, given an syntactically-valid program text, output their machine code representation, a representation which can then be executed by the CPU[61]. A defining aspect of programming languages henceforth is their theoretical *lack of ambiguity*, both in their roots in formal mathematic notation (for instance, Plankalkül was based on Frege's *Begriffschrift*) and their physical representation (punch cards are essentially discreete—hole or no hole).

Nowadays, most programming languages are Turing-complete: their syntax can implement a Turing machine and therefore simulate any possible computational aspects of any physical computer. This means that any programming language that is Turing-complete is equivalent to any other Turin-complete programming language, creating essentially a chain of equivalency between all programming languages. And yet, programming language history is full of rise and fall of languages, of hypes and disappointments, of self-claimed beautiful ones and criticized ugly ones. This is because, given such a wide, quasi-universal problem set, there are different approaches to doing so, echoing what Gilles Gaston-Granger calls

[61]The main difference between a compiler and an interpreter is that the compiler parses the whole program text as once, resulting in a binary object, while interpreters parse only one line at a time, which is then immediately executed

style, as a formal way to approach the production and communication of aesthetic, linguistic and scientific works[47]. We've already seen one difference in approaching the domain of computation: compilation vs. interpretation. Another high-level category we turn to now is that of programming paradigms.

A programming paradigm is an approach to programming based on a coherent set of principles, sometimes involving mathematical theory. Some of these concepts include hierarchy (in OOP), symbol manipulation (in functional languages), events (such as in Java) or concurrency (in Go). Each paradigm supports a set of concepts that makes it the best for a certain kind of problem[48]. These concepts in turn act as stances which infleunce how to approach, represent and prioritize all basic components that are involved in a programming language:

- **data** (what kinds of basic datatypes are built-in the language, e.g. signed integers, classes)
- **primitive operations** (how can the programmer directly operate on data, e.g. boolean logic, assignments, arithmetic operations)
- **sequence control** (how the flow of the program can be manipulated and constrained, e.g. if, while statements)
- **data control** (how the data can be initialized and assigned, e.g. type-safe vs. type-unsafe)
- **storage management** (how the programming language handles input/output pipelines)
- **operating environment** (how the program can run, e.g. virtual machine or not)

5.1.1 Theoretical qualities of programming languages

Every programming language of practical use takes a particular approach to those basic components, sometimes backed by an extended rationale (e.g. ALGOL 68), or not (e.g. JavaScript). In the case in which we are circumscribed to context-free grammars, it would be possible to optimize a particular language for an objective standard (e.g. compile time, time use, cycles used). Still, computers exist to solve problems, those problems are diverse in nature and therefore necessitate different approaches (as we've seen in our discussion of the limitations of patterns above[62]). These different approaches to what is referred to as the *problem domain* in turn influenced the development of those different paradigms, since a problem domain might have different data representations (e.g. objects, text strings, formal rules, dynamic models, etc.). Two of the early programming languages, FORTRAN and Lisp, addressed to very different problem domains: the accounting needs of businesses and the development of formal rules for artificial intelligence, respectively[63]. Some of the overarching programming paradigms are imperative (FORTRAN), functional (Lisp), object-oriented (Smalltalk) or logic (Prolog). Without diving deeper in some of the worldmaking assumptions of each of these paradigms, they are here sufficient proof to show that, while there is only one Turing-completeness, there are widely different approaches to it, some being better than others when given a specific problem.

What makes a good programming language is a matter which has been discussed amongst computer scientists, at least since the `GOTO` statement has been publicly considered harmful. Some of these discussions include both subjective arguments over preferred languages, as well as objective arguments related to performance and ease-of-use. According to Pratt and Zelkowitz:

[62] See awk as a kind of fundamentally pattern-based scripting language.

[63] For a specific discussion of these differences and how they are manifested aesthetically, see Kernighan on Pascal and C: `https://www.lysator.liu.se/c/bwk-on-pascal.html`

> The difference among programming languages are not quantitative differences in what can be done, by only qualitative differences in how elegantly, easily and effectively things can be done.[49]

Without then jumping immediately to aesthetic details of the languages they write in, programmers still express preferences (if not outright allegiances, sometimes calling themselves Pythonistas, or Rubyists). One must also keep in mind that there is a difference between considering a programming language good or beautiful *in itself*, and considered the programs written in the programming language. Turing-completeness offers an interesting challenge to the Sapir-Whorf hypothesis—if natural languages might only weakly affect the kinds of cognitive structures speakers of those languages can construct, programming languages are claimed to do so to large extents[64], even though they can all do the same thing in theory—only *how* they do it matters. These differences in the ways of doing illustrates how, in reality, different programming languages are applicable to different domains, and do so through different kinds of notations—different aesthetic features when it comes to realizing the same task.

Of the two programs presented below, the output result is exactly the same, but the aesthetic differences are obvious:

```
package main
import ”fmt”

func main() {
  var greeting = ”Hey, there.”
  fmt.Println(greeting)
}
```

[64] See Alan Perlis's Epigrams on Programming: ”A language that doesn't affect the way you think about programming, is not worth knowing.”, `https://web.archive.org/web/19990117034445/http://www-pu.informatik.uni-tuebingen.de/users/klaeren/epigrams.html`

The code above is written in Go, a language designed by Pike, Thompson and Griesemer in 2009, while the code below is written in Java, designed by James Gosling in 1995. Both are statically typed, compiled languages that are heavily influenced by C/C++ syntax, but Go is significantly younger than Java (relatively to programming languages' short history)

```
import java.io.*;

public class Greeting
{
  public static void main(String[] args)
  {
    String greeting = "Hey, there";
    System.out.println(greeting);
  }
}
```

These two snippets might seem very similar at first glance[65]From these two snippets, we can explore some of the most important criteria in programming language design: *abstraction*, *simplicity* and *orthogonality*[50], and how they underpin the writing of good programs.

Abstraction is the ability of the language to allow for the essential idea of a statement to be expressed without being encumbered by specifics which do not relate directly to the matter at hand, or to no matter at all. Abstract programming languages can lead to more succint code, and tend to hide complexity (of the machine, and of the language), from the programmer. For instance, the Java snippet above explicitly states the usage of the `System` object, in order to access its `out` attribute, and then call its `println()` method. While a lot of code here might seem verbose, or superfluous, it is in part due to it being based on an object-oriented paradigm. However, `out` object itself might seem to go particularly contrary to the requirement of programming languages to abstract out unnecessary details:

[65]For a radically different approach, see: `greeting = "Hey there"; print greeting` , a valid program in Ruby, Python and Perl, all scripting languages.

`println()` is definitely a system call dealing with I/O, and therefore already implicitly relates to the output; one shouldn't have to specify it explicitly. In contrast, Go abstracts away the system component of the print call inside the import statement `import "fmt"`. Printing, in Java, does not abstract the machine, while printing, in Go, abstracts it away in order to focus on the actual appearance of the message ("fmt" stands for "format"). Another abstraction is that of the language name itself from the import statements. When we write in Java, we (hopefully) know that we write in Java, and therefore probably assume that the default imports come from the Java ecosystem—there shouldn't be any need to explicitly redeclare it. For instance, `System.out.println()` isn't written `java.io.System.out.println()`. In contrast, the Go snippet hides the implicit `import "go/fmt"`, allow the programmer to focus, through visual clarity, on the real problem at hand, which is the logic of the program. In this direction, languages which provide more abstraction (such as Python), or which handle errors in an abstract way (such as Perl) tend to have greater readability by focusing on the most import tokens, rather than aggregating visual clutter—also called verbosity.

Related to abstraction, and a topic in itself, is the criteria of *typing*, the process of specifiying the type of a variable or of a return value (such as integer, string, etc.). A strictly-typed language such as C++ might end up being unreadable because of its verbosity, while a type-free language might be simple to read and write (at a small-scale), but dangerous to run in production. The tradeoff here is again between being explicit and safe (because a word cannot usually and intuitively be operated on in a similar way as a floating-point number), and being implicit, subtle, and dangerous (such as JavaScript's very liberal understanding of typing). With experience, typing can usually be inferred by purely aesthetic means: Python's boolean values are capitalized (`True, False`) and its difference between string and byte is represented by the use of double-quotes for the former and single-quotes for the latter. In the case above, explicitly having to mention that `greeting` is of type `String` is again redundant, since it is already hinted at

by the double-quotes (and, indeed, single quotes are byte types in Java as well). Go doesn't force programmers to explicitly declare variable types (they can, if they want to), but in this case they let the computers do the heavy lifting of specifying something that is already obvious to the programmer.

A particularly note-worthy example of an elegant solution to the tradeoff between type safety and readability can be found in Go's handling of error values returned by functions:

```
d, _ := exec.LookPath("date")
fmt.Println("Today is %s", d)
```

The _ character which we see on the first line is the choice made by Go's designers to force the user to both acknowledge and ignore the potential error value that is returned by executing the external command. This particular character, acting as an empty line, *represents absence*, not cluttering the layout of the source, while reminding subtly of the *potential* of this particular statement to go wrong and crash the program. Abstraction is therefore a tradeoff between explicitly highlighting the computer concern (how to operate practically on some data or statement), and hiding anything but the human concern (whether or not that operation is of immediate concern to the problem at hand at all). As such, languages who offer powerful abstractions tend not to stand in the way of the thinking process of the programmer. This particular example of the way in which Go deals with error-handling is a great example of the designer's explicit stylistic choice. In the words of Niklaus Wirth:

> Stylistic arguments may appear to many as irrelevant in a technical environment, because they seem to be merely a matter of taste. I oppose this view, and on the contrary claim that stylistic elements are the most visible parts of a language. They mirror the mind and spirit of the designer very directly, and they are reflected in every program written.[51]

Orthogonality, in turn, relates to the ability of a small set of simple syntactics constructs to be recombined in order to achieve greater complexity, while remaining independent from each other. A direct consequence of such a feature is the ease with which the programmer can familiarize themselves with the number of constructs in the language, and therefore their ease in using them without resorting to the language's reference. Relating back to Ricoeur, the orthogonality of a language offers a simple but powerful solution to the polysemy of the word (here, token) as embedded within a broader sentence (here, statement). The expressivity of a statement comes not just from the individual keywords, but rather from their combination. For instance, the example of Lisp treats both data and functions in a similar way, essentially allowing the same construct to be recombined in powerful ways (again, the Lisp interpreter comes to mind), while the Ruby language, and its foundational design choice which makes every type (themselves abstracted away) an object allows for greater creativity, through familiarity, in writing code, making the language itself more habitable. Orthogonality also implies independence, since all constructs operate distinctly from each other, while remaining related, because in cooperation with each other. This offers a solution to the cognitive burden of the *non-atomicity* of computer programs, in which data can end up being tangled in a non-linear program execution, and become unreadable. This unreadability is triggered, not by verbosity, but because of the uncertainty of, and confusion about, the potential side-effects caused by any statement. Such independence in all constructs in turn presents a kind *symmetry*, itself is a well-accepted aesthetic feature of any artefact, in that the use of each of the constructs is similar. This similarity eases the cognitive friction in writing and reading code since an orthogonal language allows the programmer to rely on the fact that, deep down, everything is the same, and not a collection of quirks and arbitrary decisions. For example, the below C code is illegal:

```
int[] getListOfPrimeNumbers(){
  //-- this function is illegal!
}
```

The above code is a specific instance of one of those quirks: the fact that C cannot return arrays from functions requires both a deep knowledge of the language implementation and a willingness to accept that this is how things are, even though other languages allow for such a feature. In this case, the language exhibits an un-orthogonal property since the two constructs (`return` and `int[]`) interact with each other in non-independent ways.

Finally, one of the consequences of such a feature is the shift from computer semantic interpretation (usually connected to strongly-typed languages) to human-interpretation (and weakly-typed languages). Non-orthogonality implies that the compiler (as a procedural representation of the language) has the final say in what can be expressed, while orthogonal languages leave more leeway to the writer in keeping track of which value and statement does what, allowing for both more creativity and more uncertainty in the interpretation and execution of the program.

Both of these features, abstraction and orthogonality, ultimately relate to simplicity:

> Simplicity enters in four guises: uniformity (rules are few and simple), generality (a small number of general functions provide as special cases a host of more specialized functions, orthogonality), familiarity (familiar symbols and usages are adopted whenever possible), and brevity (economy of expression is sought).[52]

The point of a simple programming language is a programming language which does not stand in the way of the program being written, or of the problem being addressed. Such a goal is achieved in part by having accurate conceptual mappings between computer expression mapping and

human mapping (such as the code block-human sentence mapping[53]) If one is to write a program related to an interactive fiction in which sentences are being input and output in C, then the apparently simple data structure `char` of the language reveals itself to be cumbersone and complex when each word and the sentence that the programmer wants to deal with must be present not as words, but as series of `char` (hence the origin of the name of the data type `string`, as a continuous series of `char`). As we've seen, a simple language does not mean that it is easy (perhaps the simplest language of all being lambda-calculus, is far from an easy construct to grasp), but that it is just a means to an end, akin to any other tool or instrument[66].

Programming languages, however, are *symbolic* tools, manipulating *symbolic* matter. As a formal system of symbols which can sustain creations with aesthetic features, programming languages do share commonalities with Goodman's *Languages of Art*.

Goodman develops in his opus a systematic approach to symbols in art, freed from any media-specificity (from pictorial symbols to musical notations and even time marks on clocks and watches). With it, he accomplished two things: he highlights the ways in which symbols systems have expressive and communicative power (through the dyads of denotation and exemplification, description and representation, possession and expression), and what are the kinds of requirement that such a system must have in order to develop these expressive and communicative abilities. These requirements are that of unambiguity, syntactic and semantic disjointedness, and differentiation[54]. Looking at programming languages from this perspective argues for their communicative and expressive power. From the perspective of the computer, programming languages are unambiguous insofar as any expression or statement will ultimately result in an unambiguous result by the CPU (if any ambiguity remains, the program crashes). They are also syntactically and semantically disjointed (i.e. clearly distin-

[66]For a further parallel on musical instruments, see Rich Hickey's keynote address at RailsConf 2012: `https://www.youtube.com/watch?v=rI8tNMsozo0`

guishable from one another). The use of formal notations, such as BNF, had for aim to resolve any possible ambiguity in the syntax of the language in a very clear fashion. The semantics of programming languages, as we will see below, also aim at being thoroughly disjointed: a variable cannot be of multiple types at the same time. Finally, programming languages are also differentiated systems since no symbol can refer to two things at the same time.

The tension arises when it comes to the criteria of unambiguity, from a human perspective. The most natural-language-like component of programs, the variable and function names, always have the potential of being ambiguous (e.g. does `int numberOfFlowers` refer to the current number of flowers in memory? To the total number of potential of flowers? To a specific kind of number whose denomination is that of a flower?). We consider this ambiguity a productive opportunity for creativity, and a hindrance for program effectiveness. So, given the qualification of programming languages as symbolic systems, we could expand our short analysis above by inspecting how programming languages allow for program texts which denote, label, represent, etc. in order to further argument how source code has the potential, and has examples, of being an artistic mean of expression and comprehension, from a cognitive point of view.

If they are aesthetic symbol systems, then they can also elicit emotional responses. As we've seen with software patterns, what also matters to programming languages is not just their design, but their *situated* use:

> Before closing, let me mention another essential ingredient, one that hardly ever gets mentioned: It must be a pleasure and a joy to work with a language, at least for the orderly mind. The language is the primary, daily tool. If the programmer cannot love his tool, he cannot love his work, and he cannot identify himself with it.[51]

Bringing it back to architecture, the language designer bruce McLennan

further presses the point:

> There are other reasons that elegance is relevant to a well-engineered programming language. The programming language is something the professional programmer will live with - even live in. It should feel comfortable and safe, like a well-designed home or office; in this way it can contribute to the quality of the activities that take place within it. Would you work better in an oriental garden or a sweatshop?[55]

5.1.2 Practical qualities of programming languages

Concrete use of programming languages operate on a different level of formality: if programming paradigms are top-down strategies specified by the language designers, they find their mirror in the bottom-up practices of softare developers (to borrow Michel De Certeau's terminology). Such practices crystallize, for instance, in *idiomatic writing*. Idiomaticity refers, in traditional linguistics, to the *realized* way in which a given language is used, in contrast with its possible, syntactically-correct and semantically-equivalent, alternatives. For instance, it is idiomatic to say "The hungry dog" in English, but not "The hungered dog" (a correct sentence, whose equivalent is idiomatic in French and German). It therefore refers to the way in which a language is a social, experiential construct, relying on intersubjective communication[56]. Idiomaticity is therefore not a purely theoretical feature, but first and foremost a social one. This social component in programming languages is often reliant on knowledge of said language, and of its quirks. In this sense, programming language communities are akin to hobbyists clubs, with their meetups, mascots, conferences and inside-jokes[67].

[67] For an example of such joke, see Gary Bernhardt's talk on JavaScript: `https://www.destroyallsoftware.com/talks/wat`

So an idiom in a programming language depends on the human interpretation of the formal programming paradigms (since, in most programming languages today and especially in scripting languages, paradigms are blended and no language is purely single-paradigmatic). Such an interpretation is also manifested in community-created and community-owned documents, such as *The Zen of Python*[68].

The Zen of Python shows how the philosophy of a programming language relates to the practice of programming in it. Without particular explicit directives, it nonetheless highlights *attitudes* that one should keep in mind and exhibit when writing Python code. Such a document sets the mood and the priorities of the Python community at large (being included in its official guidelines in 2004), and highlights a very perspective on the priorities of theoretical language design. For instance, the first Zen is:

> Beautiful is better than ugly.

An obvious statement which prompts non-obvious questions (how do I write beautiful code? Can I really tell if my code is ugly?), this epigram sets the focus on a specific aspect of the code, rather than on a specific implementation. With such broad statements, it also contributes to strengthening the community bonds by creating shared, folk knowledge. In practice, writing idiomatic code requires not only the awareness of the community standards around such an idiomaticity, but also knowledge of the language construct themselves which differentiate it from different programming languages. For instance, in Python:

```
for i in range [0, 1, 2, 3, 4, 5]:
  print i
```

is semantically equivalent to:

```
for i in range(5):
  print i
```

[68] Tim Peters, 1999: `https://docs.python-guide.org/writing/style/#zen-of-python`

but only the second example is considered idiomatic Python, partly because it is *specific* to Python, and because more performing than the first example, due to the desire of the developers of Python to encourage idiomaticity (i.e. what they consider good Python to be). Beautiful code, then seems to be a function of knowledge, not just of what the intent of the programmer is, but knowledfge of the language itself as a differentiated idiom. Another example[69] of beautiful, because idiomatic, Python code is:

```
@lru_cache(3)
def fib(n):
  return n if n < 2 else fib(n - 1) + fib(n - 2)
```

This function calculates the Fibonacci sequence (a classic exercise in computer programming), but makes a clever use of decorators in Python. The `@lru_cache(3)` line caches the last 3 results in the least-recently used order, closely mirroring the fact that the Fibonacci sequence only ever needs to compute the terms n, n-1 and n-2, thus closely aligning the language domain and the problem-domain. Through this, the programmer uses a key, advanced feature of the language in order to make the final program more terse, more precise, and more closely aligned with the problem domain than other implementations, to the detriment of a decrease in readability for non-Pythonistas.

Idiomaticity reflects what the *aesthetic intent* of the language designers and implementers is. Notation matters, and designers want to encourage good practices through good notations, assuming that programmers would gravitate towards what is both the most efficient and the best-looking solution. For example, one of the biggest differences between object-oriented and non-object-oriented programming is the possibility to identify the actor of an action using purely syntactic means[57]. Another way to encourage writing good code is through the addition of *syntactic sugar*. Syntactic sugar describes the aesthetic features of the language who are variants

[69] From `https://www.quora.com/What-makes-some-code-beautiful`

of a similar feature, and where the only difference between them is their appearance—i.e. visual, semantic shortcuts. The looping examples above are good instances of syntactic sugar, albeit with performance differences. The Ruby language is riddled with syntactic sugar, and highlights how syntactic sugar can "sweeten" the reading process, aiming for more clarity, conciseness, and proximity to natural languages. In Ruby, to access a boolean value on an attribute of an object, one would write:

```
if Being.alive
  puts "and well"
```

The syntactic sugar comes in the form of the question mark:

```
if Being.alive?
  puts "and well"
```

There is absolutely no functional differences in the statements above, and the question mark is just here to make the code seem more natural and intuitive to humans. Checking for a boolean (or non-nil value) in an if statement is, in the end, the equivalent of asking a question about that value Here, Ruby makes that explicit, therefore making it easier to read with the most minimal amount of additional visual noise (i.e. one character).

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

Since programming languages are meant to help programmers solve se-

mantic issues (in the problem domain) through elegant syntactical means, and since they act as interfaces between the programmer and the machine, we now turn to the question of how semantics are represented programmatically, and whether machine understanding exists.

5.2 Programming Semantics

On of the reasonings behind the formal approach to programming languages, besides the very material machine requirements of a circuit design based on discreete distinctions, is, according to the designers of ALGOL 58, the dissatisfaction with the fact that subtle semantic questions remained unanswered due to a lack of clear description[58]. If the goal of a program text is to be syntactically and semantically clear, and if programming languages are syntactically unambiguous, we examine here under what form do semantics exist as computer representations, and what kind of specific semantic issues are at stake when writing program texts. The very requirement for semantic representation in program language design is first and foremost due to the fact that:

> The first and most obvious point is that whenever someone writes a program, it is a program about something.[20]

A statement which is itself followed by the tension between semantics and syntax. Semantics have the properties of aboutness and directedness (they point towards something external to them), and syntax has the property of (local) consistency and combination (they function as a mostly closed system). Looking at programing languages as *applied* mathematics, in the sense that it is the art and science of manipulating formal tokens, tokens which in turn represent elements in the world of some kind, we arrive at the issue of defining semantics (meaning) in strictly computer-

understandable terms.

Meaning is created by an active reading, in which the linguistic form enables interpretation, rather than exclusively conveying information. Winograd states that interpretation happens through *grounding*, essentially contextualizing information in order to interpret it and extract meaning. He identifies three different kinds of grounding. The e*xperiential* grounding, in which verification is made by direct observation, related to the role of the senses in the constitution of the conceptual structures that enable our understanding of the world—also known as the material implementation of knowledge. The *formal* grounding relies on logical and logical statements to deduce meaning from previous, given statements that are known. Finally, *social* grounding relies on a community of individuals sharing similar conceptual structures in order to qualify for meaning to be confirmed. Of these three groundings, programming languages rely on the second.

The reason for the bypassing of experiential and community grounding can be found in one of the foundations of computer science, as well as information science: Claude Shannon's mathematical theory of communication. In it, he postulates the separation of meaning from information, making only the distinction between signal and noise. Only formal manipulation of signal can then reconstitute meaning[70]. Indeed, according to Brian Cantwell-Smith, computing is *meaning mechanically realized*, due to the fact that the machine comes from non-mechanical origins[71]. We think of computers as digital but they can be seen as only the digital implementation of the phenomenon of computation, with its roots in formal logic. It is therefore through formal logic that one can recreate meaning through the exclusive use of the computer.

A computer is actually a collection of layers, each defining different levels of machines, with different semantic capabilities. First, it is a physical

[70] An affordance that is shared with literature, according to Peter Suber[59]

[71] Retrieved from: `https://web.archive.org/web/20160826234606/http://ageofsignificance.org/aos/en/aos-v1c0.html`

machine, dealing with voltage differences. These voltage differences are then quantized into binary symbols, in order to become manipulable by a logical machine. From this logical machine is built an abstract machine, which uses logical grounding in order to execute specific, pre-determined commands. The interpretation of which commands to execute, however, leaves no room for the kind of semantic room for error that humans exhibits (particularly in hermeneutics). It is a strictly defined mapping of an input to an output, whose first manifestation can be found in the symbols table in Turing's seminal paper. The abstract machine, in turn, allows for high-level machines (or, more precisely, high-level languages which can implement any other abstract machine). These languages themselves have linguistic constructs which allow the development of representational schemes for data (i.e. data structures such as `structs`, `lists`, `tuples`, `objects`, etc.). Finally, the last frontier, so to speak, is the subject domain: the things that the programmer is talking about. These are then represented in data structures, manipulated through high-level languages, processed by an abstract machine and executed by a logical machine which turns these representations into voltage variations.

The subject domain is akin to a semantic domain, a specific conceptual place that shares a set of meanings, or a language that holds its meaning, within the given context of this place. And there is only one context which the computer provides: itself. Within this unique context, semantics still hold a place in any programming language textbook, and is addressed regularly in programming language research. Concretely, *semantics in computer programming focuses on how variables and functions should behave*[58]. Given the statement `l := j + p`, the goal of programming language semantics is to deduce what is the correct way to process such a statement; there will be different ways to do so depending on the value and the type of the `j` and `p` variables. If they are strings, then the value of `j` will be their concatenation. If they are numbers, it will be their addition, and so on.

This problem is called the *use-mention* problem, which requires the reconciliation of the name of entities, tokens in source code, with the entities themselves, composed of a value and a type. The way this is achieved is actually quite similar to how syntax is dealt with. The compiler (or interpreter), after lexical analysis, constructs an abstract syntax tree representation of the statement, separating it, in the above case, in the tokens: `l`, `:=`, `j`, `+` and `p`. Among these, `:=` and `+` are considered terminal nodes, or leaves, while the other values still need to be determined. The second pass represents a second abstract syntax tree through a so-called semantic analysis, which then *decorates* the first tree, assigning specific values (attributes) and types to the non-terminal nodes, given the working environment (e.g. production, development, test). This process is called *binding*, as it associates (binds) the name of a variable with its value and its type. Semantics is thus the decoration of parsed ASTs, evaluating attribute—which can be either synthesized or inherited. Since decoration is the addition of a new layer (a semantic layer) on top of a base layer (a syntactic one), but of a similar tree form, this leads to the use of what can be described as a *meta-syntax tree*.

In terms when the values are being bound, there are multiple different binding times, such as language design time (when the meaning of `+` is defined), compile time, linker time, and programming writing time. It is only during the last one of these times, that the programmer inserts their interpretation of a particular meaning (e.g. `j := "jouer"`, meaning one of the four possible actions to be taken from the start screen of a hypothetical video game). Such a specific meaning is then shadowed by its literal representation (the five consecutive characters which form the string) and its pre-defined type (`strings`, here in Go). This process does show that the meaning of a formal expression can, with significant difficulty and clumsiness, nonetheless be explained; but the conceptual content still eludes the computer, varying from the mundane (e.g. a simple counter) to the almost-esoteric (e.g. a playful activity). Even the most human-beautiful

code cannot force the computer to deal with new environments, in which meaning has, imperceptibly, changed. Indeed,

> In programming languages, variables are truly variable, whereas variables in mathematics are actually constant[51].

From this perspective, the only thing that the computer does know that the programmer doesn't, and which would "make its life easier", the same way that the programmer's life can be made easier through beautiful code, is how the code is represented in an AST, and where in physical memory is located the data required to give meaning to that tree[52]. We might hypothesize that beautiful code, from the computer's perspective, is code which is tailored to its physical architecture, a feat which might only be realistically available when writing in Assembly[72]. Before we turn to how such a code is written by the particular group of humans referred to as hackers, there are nevertheless some concepts in programming which do not have simple meaning for humans, re-iterating the need of aesthetics to make these concepts graspable.

5.3 Idiosyncracies of computing-specific constructs

Computation, in its philosophical sense, is a complex and debated concept[17, 60]. In short, software isn't simple, it's a cognitive artefact which can be understood at the physical, design and intentional levels[61]. With odern programming languages allowing us to safely ignore the first level, it is at the interaction of the design (programming) and intentional (human) level that things get complicated; the question "what does a Turing machine do?" has `n+1` answers, `1` syntactic answer, and `n` semantic ones, based on however many interpretations.

[72] For a mythical telling of such a process, see the story of Mel, A Real Programmer: `https://www.cs.utah.edu/~elb/folklore/mel.html`.

Without diving into the depths of the philosophy of computation, we highlight two programming concepts which tend to be evident to the computer (evident in the sense some see them as emergent properties of computation), and yet quite complex to deal with for humans: *referencing* and *threading*.

Referencing is a surface-level consequence of the *use-mention* problem referred to above, the separation between a name and its value, with the two being bound together by the address of the physical location in memory. As somewhat independent entities, it is possible to manipulate them separately, with consequences that are not intuitive to grasp. Some programming languages allow for this direct manipulation, through something called *pointer arithmetic*[73]. Indeed, the possibility to add and substract memory locations independent of the values held in these locations, as well as the ability to do arithmetic operations between an address and its value isn't a process whose meaning comes from a purely experiential or social perspective, but rather exists meaningfully for humans only through logical grounding, by understanding the theoretical architecture of the computer. What also transpires from these operations is another dimension of the non-linearity of programming languages, demanding complex mental models to be constructed and updated to anticipate what the program will ultimately result in when executed. Notation attempts at remediating those issues by offering symbols to represent these differences, such as:

```
int date = 2046; // ‘date‘ refers to the literal value of the number
    2046
int *pointer = &date; // ‘pointer‘ refers to the address where the
    value of ‘date‘ is stored, e.g. 0x5621
*pointer = 1996; // this accesses the value located at the memory
    address held by ‘pointer‘ (0x5621) and sets it to 1996
std::cout << date; // prints the literal value of date, at the
    address 0x5621: 1996
```

[73] For better or worse, C is very liberal with what can be done with pointers.

The characters `*` and `&` are used to signal that one is dealing with a variable of type pointer, and that one is accessing the pointed location of a variable, respectively. Line 2 of the snippet above is an expression called *dereferencing*, a neologism which is perhaps indicative of the lack of existing words for referring to that concept. In turns, this hints at a lack of conventional conceptual structures to which we can map such a phenomenon.

Threading is the ability to do multiple things at the same time. While the concept itself is simple, to the point that we take it for granted in modern computer applications since the advent of time-sharing systems. However, the proper handling of threading when writing and reading software is itself complex. This involves the ability to demultiply the behaviour of routines (already non-linear) to keep track of what *could* be going on at any point in the execution of the program, including use and modification of shared resources, the scheduling of thread start and end, as well as synchronization of race conditions (e.g. if two things happen at the same time, which one happens first, such that the consistence of the global state is preserved?). As Edward A. Lee put it:

> Although threads seem to be a small step from sequential computation, in fact, they represent a huge step. They discard the most essential and appealing properties of sequential computation: understandability, predictability, and determinism. Threads, as a model of computation, are wildly nondeterministic, and the job of the programmer becomes one of pruning that nondeterminism[62].

Threading shows how the complexity of a deep-structure needs to be adequately represented in the surface. Once again, aesthetically-satisfying (simple, concise, expressive) notation can help programmers in understanding what is going on in a multi-threaded program, by removing additional cognitive overload generated by verbosity[74]. Here are two of the

[74] For an example of code that looks good on the surface, but is deeply wrong, see

simplest examples, in C and in Go:

p.105 of `https://mirrors.edge.kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook-e2.pdf`.

```
  #include <iostream>
  #include <thread>

  void recall(int date)
  {
      std::cout << date << '\n';
  }

  int main()
  {
    std::thread thread(recall, 2046);

    thread.join();

    cout << "We're done!";

    return 0;
}
```

```
  package main

  import (
      "fmt"
  )

  func recall(int date) {
      fmt.Println(date)
  }
  func main() {
      go recall()

      fmt.Println("We're done!")
  }
```

Once again, we see how the abstraction provided by some language constructs in Go result in a simpler and more expressive program text. In this case, the non-essential properties of the thread are abstracted away

from programmer concern. The *double-meaning* embedded in the `go` keyword even uses a sensual evokation of moving away (from the main thread) in order to stimulate implicit understanding of what is going on.

In conclusion, programming languages, as aesthetic symbol systems, are essential in allowing for aesthetic properties to emerge during the writing process of program texts. They present affordances for the abstraction and combination of otherwise-complex programming concepts, for the development of familiarity through their idiomatic uses and for ease of readability—to the point that it might become transparent to experienced readers. Still, since understanding is at stake, there is a kind of beautiful code which bypasses any semantic representation embedded in these languages by speaking directly to the machine. This is the kind of beauty which we look at in the following section.

---

6 Aesthetics of hacks

Along with software developers and source code poets, another community of program writers are the hackers. While there is no absolute, uncrossable boundary between each of these communities, they still differentiate themselves mainly in terms of *purpose* and in terms of *means*. By examining the purpose of the code written by hackers, and the way in which it is written, we highlight yet another perspective on the aesthetics of source code, and how it intersects once more with understanding, architecture, and skill.

To do so, we first disambiguate the term to clarify these purposes and means. Then, we examine specific examples of software written by hackers, from the one-liner to the UNIX operating system, with a detour through an esoteric programming language and a demoscene program. These will provide an empirical basis to touch upon the concepts of elegance and unscrutability in writing and reading code.

6.1 Principles of hackers

Often referred to as malicious computer manipulators by popular media, the term hacker nonetheless refers to a more specific (and less harmful) part of the computer programmer group. In short, hackers are individuals whose priorities can be summed up in the *hacker ethic*, formalized in Stephen Levy's history of the community[63]. This set of principles describes the outlook that hackers have on information, authority, beauty and skill, all seen through the lens of the computer. Of all of these, *skill* is perhaps the most defining aspect. The earnest acquisition and execution of computer-related skills is a crucial requirement for being recognized as a hacker, and these positions vis-à-vis authority seem to be taken in most part to support such an approach.

Information wants to be free provides the fertile epistemic environment for learning more about any topic. This stance is particularly highlighted in the hacker perspective on copyright, emphasizing the difference between *free* software, as in free of charge, and *free* software, as in liberated from any restrictions in its diffusion, modification and replication. In this context, code is considered as liberated knowledge, rather than as a constrained commodity, allowing other hackers to inspect and learn from any freely distributed software. This highlights a material stance of computer know-how: only through direct engagement can one really know something—hackers learn by doing, by tinkering, sometimes breaking systems, and considering these breaks as contributions towards greater understanding of the system they're dealing with.

Similarly, any authority or credential system which isn't based on practical know-how is distrusted: merit is based on what one can do alone, without regards for race, gender or class[75]. It is what one can do (most often with a technological system), which determines their value to the eyes of the hacker community, and not what degrees they obtain[76]. For instance, for hackers, the "true hero" of the Apple-led computer revolution isn't Steve Jobs, the product manager, but rather Steve "Woz" Wozniak, the engineer and developer behind the Apple I[77], as well as the infamous *blue boxes*. For hackers, what is claimed to be known is only relevant insofar as it can be actually done, and preferably well. Knowledge is therefore a prerequisite for excellence, but isn't sufficient. Indeed, another dimension is *how* it should be done: to hackers, it matters little what a source of authority tells them they can or cannot do with a particular system. Furthermore,

[75]Incidentally, most of the well-known hackers are overwhelmingly white and affluent, even though there is anecdotal evidence of inclusivity for the LGBTQ+ communities in hacker groups.

[76]Again, most of the well-known hackers of that community gravitated around elite US universities.

[77]For instance, see Wire's portrait here: `https://www.wired.com/2006/02/forget-jobs-lets-worship-woz/`

whether it can be done informs the engagement in these systems far more than whether or not it is allowed, or if it should be done. This hierarchy of priorities is perhaps also the reason why hackers became associated in the public consciousness as malevolent actors, since technical exploits can sometimes straddle the line of what is considered to be legal[78].

Finally, hackers believe that *you can create art and beauty on a computer*, but such a statement is to be understood with an emphasis on the process itself, rather than the result. Phreaking, a phenomenon immediately predating computer hacking, offers a particularly telling example of how the means are valued more than the ends. John Draper, also known as Cap'n Crunch, adapted this moniker because the Captain Crunch cereal boxes included, for a while, a whistle which could generate a specific 2600KHz tone, which could in turn manipulate the routing of phone lines. He then used such a whistle from a phone booth to dial from operator to operator accross the world, in order to reach, in the end, the very phone booth he was calling from—only to hear a busy tone[64]. Hackers can be known for finding unexpected solutions to complex problems with very little concrete practical use, except to prove that it can be done, and can be done *cleverly*.

This last point highlights an ambiguity in the hacker attitude: with undertones of anarchism, neo-libertarianism and hippie culture, they appear to both strive for a better society but exhibit at the same time a radical propensity to interact with a computer *for its own sake*. The attention given to the technical feat sometimes overshadows the practical use, as we've seen above, and the direct social consequences, focusing rather on the eminently self-sufficient entertainment value of interacting in clever ways with a computer—as stated by Linus Torvalds[65]. Witness to some of the early hackers in the 1960s at MIT, Weizenbaum describes them as such:

[78] *Aaron Swartz vs. United States* represents a particular case of the overlap between the hacking and legal frameworks.

> To hack is, according to the dictionary, ”to cut irregularly, without skill or definite purpose; to mangle by or as if by repeated strokes of a cutting instrument”. I have already said that the compulsive programmer, or hacker as he calls himself, is usually a superb technician. It seems therefore that he is not ”without skill” as the definition will have it. But the definition fits in the deeper sense that the hacker is ”without definite purpose”: he cannot set before him a clearly defined long-term goal and a plan for achieving it, for he has only technique, not knowledge. He has nothing he can analyze or synthesize; in short, he has nothing to form theories about. His skill is therefore aimless, even disembodied. It is simply not connected with anything other than the instrument on which it may be exercised. His skill is that of a monastic copyist who, though illiterate, is a first rate calligrapher.[66]

While he looks down on hackers, perhaps unfairly, from the perspective of a computer scientist whose theoretical work can be achieved only through thought, pen and paper, the point still remains: hackers are first and foremost technical experts who can get lost into technics for their own sake. From a broad perspective, hackers therefore seem to exhibit an attitude of *direct engagement*, *subverted use* and *technical excellence*. We now look at some instances of hacker source code to identify how such an approach in manifested concretely.

6.2 One-liners

The *one-liner* is a piece of source code which fits on one line, and is usually intepreted immediately by the operating system. They are terse, concise, and eminently functional: they accomplish one task, and one task only. This binary requirement of functionality (in the strict sense of: ”does it do

what it's supposed to do, or not?") actually finds a parallel in a different kind of one-liners, the humoristic ones in jokes and stand-up comedy. In this context, the one-liner also exhibits the features of conciseness and impact, with the setup conflated with the punch line, within the same sentence. One-liners are therefore self-contained, whole semantic statements which, through this syntactic compression, appear to be clever—in a similar way that a good joke is labelled clever.

In programming, one-liners have their roots in the philosophy of the UNIX operating system, as well as in the early diffusion of computer programs for personal computer hobbyists[67]. On the one side, the Unix philosophy is fundamentally about building simple tools, which all do one thing well, in order to manipulate text streams[68]. Each of these tools can then be piped (directing one output of a program-tool into the input of the next program-tool) in order to produce complex results—reminiscing of the orthogonality feature of programming languages. Sometimes openly acknowledged by language designers—such as those of AWK—the goal is to write short programs which shouldn't be longer than one line. Given that constraint, a hacker's response would then be: how short can you make it?

If writing one-line programs is within the reach of any medium-skilled programmer, writing the shortest of all programs does become a matter of skill, coupled with a compulsivity to reach the most syntactically compressed version. For instance, Guy Steele[79] recalls:

> This may seem like a terrible waste of my effort, but one of the most satisfying moments of my career was when I realized that I had found a way to shave one word off an 11-word program that [Bill] Gosper had written. It was at the expense of a very small amount of execution time, measured in fractions of a machine cycle, but I actually found a way to shorten his code by 1 word

[79] Influential langugage designer, who worked on Scheme, ECMAScript and Java, among others.

and it had only taken me 20 years to do it[69].

This sort of compulsive behaviour is also manifested in the practice of *code golf*, challenges in which programmers must solve problems by using the least possible amount of characters—here, the equivalent of *par* in golf would be Kolmogorov complexity[80]. So minimizing program length in relation to the problem complexity is a definite feature of one-liners, since choosing the right programming language for the right taks can lead to a drastic reduction of syntax, while keeping the same expressive and effective power. Tasked with parsing a text file to find which lines had a numerical value greater than 6, Brian Kernighan writes the following code in C[81]:

[80] See: `https://en.wikipedia.org/wiki/Kolmogorov_complexity`

[81] From Succesful Language Design, Brian Kernighan at the University of Nottingham, `https://www.youtube.com/watch?v=Sg4U4r_AgJU`

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

int main(void){
  char line[1000], line2[1000];
  char *p;
  double mag;

  while(fgets(line, sizeof(line), stdin) != NULL) {
    strcpy(line2, line);
    p = strtok(line, "\t");
    p = strtok(NULL, "\t");
    p = strtok(NULL, "\t");
    sscanf(p, "%lf", &mag);
    if(mag > 6) /* $3 > 6 */
      printf("%s", line2);
  }

  return 0
}
```

The equivalent in AWK, a language he designed, and which he actually refers to in the comment on line 15, presumably as a heuristic as he is writing the function, is:

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

The difference is obvious, not just in terms of formal clarity and cleanliness of the surface structure, but also in terms of matching the problem domain: this obviously prints every line in which the third field is greater than 6. The AWK one-liner is more efficient, more understandable because more intuitive, and therefore more beautiful. On the other hand, however, one-liners can be so condensed that they loose all sense of clarity for someone who doesn't have a deep knowledge in the specific language in which it is written. Here is Conway's game of life implemented in one line of APL:

```
life ← {⊃1 ⍵ ∨.∧ 3 4 = +/ +⌿ ¯1 0 1 ∘.⊖ ¯1 0 1 ⌽¨ ⊂⍵}
```

The obscurity of such a line—due to its highly-unusual character notation, and despite the pre-existing knowledge of the expected output—shows why one-liners are usually highly discouraged for any sort of code which needs to be *worked on* by other programmers. Cleverness in programming indeed tends to be seen as a display of the relationship between the programmer and the machine, rather than between different programmers, and only tangentially about the machine. On the other hand, though, the nature of one-liners makes them highly portable and shareable, infusing them with what one could call *social beauty*. Popular with early personal computer adopters, at a time during which the source code of programs were printed in hobbyist magazines and needed to be input by hand, and during which the potential of computation wasn't as widely distributed amongst society, being able to type just one line in, say, a BASIC interpreter, and resulting in unexpected graphical patterns created a sense of magic and wonder in first-time users—how can so little do so much?[82].

Another example of beautiful code written by hackers is the UNIX operating system, whose inception was an informal side-project spearheaded by Ken Thompson and Dennis Ritchie in the 1970s. As the first portable operating system, UNIX's influence in modern computing was significant, e.g. in showing the viability and efficiency of text-based processing, hierarchical file-system, shell scripting and regular expressions, amongst others. UNIX is also one of the few pieces of production software which has been carefully studied and documented by other developers. One of the most famous examples is *Lions' Commentary on UNIX 6th Edition, with Source Code* by John Lions, an annotated edition of the UNIX source code, which was circulated illegaly in classrooms for twenty years before its official publication was authorized by the copyright owners[70]. Coming back to the relationship between architecture and software development, Christopher Alexander asks, in the preface of Richard P. Gabriel's *Patterns of Soft-*

[82]For an example fo such one-liner, see for instance: `https://www.youtube.com/watch?v=0yKwJJw6Abs`

ware[44],

> *For a programmer, what is a comparable goal? What is the Chartres of programming? What task is at a high enough level to inspire people writing programs, to reach for the stars?*

And UNIX might be one of the answers to that question, both by its functionality, and by its conciseness, if not alone by its availability. Another program which qualifies as beautiful hacker code, due both to its technical excellence, unusual solution and open-source availability is the function to compute the inverse square root of a number, a calculation that is particularly necessary in any kind of rendering application (which heavily involves vector arithmetic). It was found in the *Quake* source code, listed here verbatim[83]:

[83] The Quake developers aren't the authors of that function, the merit of which goes to Greg Walsh, but are very much the authors of the comments.

```
float Q_rsqrt( float number )
{
    long i;
    float x2, y;
    const float threehalfs = 1.5F;

    x2 = number * 0.5F;
    y  = number;
    i  = * ( long * ) &y;    // evil floating point bit level hacking
    i  = 0x5f3759df - ( i >> 1 );               // what the fuck?
    y  = * ( float * ) &i;
    y  = y * ( threehalfs - ( x2 * y * y ) );   // 1st iteration
//  y  = y * ( threehalfs - ( x2 * y * y ) );   // 2nd iteration,
                                                // this can be removed

    return y;
}
```

What we see here is indeed a combination of the understanding of the problem domain (what's the acceptable result I need to maintain a high-framerate with complex graphics), and how the specific knowledge of computers (i.e. bit-shifting of a float cast as an integer) and the snappiness and wonder of the the comments (*what the fuck?* indeed). The use of `0x5f3759df` is what programmers call a *magic number*, a literal value whose role in the code isn't made clearer by a descriptive variable name. Usually bad practice and highly-discouraged, the magic number here is exactly that: it does makes the magic happen.

Further examples of such intimate knowledge of both the language and the machine can be found in the works of the *demoscene*. Starting in Europe in the 1980s, demos were first short audio-visual programs which were distributed along with *crackware* (pirated software), and to which the names of the people having cracked the software were prepended, in the form of a short animation[71]. Due to this very concrete constraint—there was only so much memory left on a pirated disk to fit such a demo—

programmers had to work with these limitations in order to produce the most awe-inspiring effects. Indeed, one notable feature of the demoscene is that the output should be as impressive as possible, as an immediate, phenomenological appreciation of the code which could make this happen[84]. Indeed, the `comp.sys.ibm.pc.demos` news group states in their FAQ:

> A Demo is a program that displays a sound, music, and light show, usually in 3D. Demos are very fun to watch, because they seemingly do things that aren't possible on the machine they were programmed on.
>
> Essentially, demos "show off". They do so in usually one, two, or all three of three following methods:
>
> - They show off the computer's hardware abilities (3D objects, multi-channel sound, etc.)
> - They show off the creative abilities of the demo group (artists, musicians)
> - They show off the programmer's abilities (fast 3D shaded polygons, complex motion, etc.)[**?**]

This showing off, however, does not happen through immediate engagement with the code from the reader's part, but rather in the thorough explanation of the minute functionalities of the demo by its writer. Because of these constraints of size, the demos are usually written in C, openGL, Assembly, or the native language of the targeted hardware. Source code listings of demos also make extensive use of shortcuts and tricks, and little attention is paid to whether or not other humans would directly read the source—the only intended recipient is a very specific machine (e.g. Commodore 64, Amiga VCS, etc.). The release of demos, usually in demoparties, are sometimes accompanied by documentation, write-ups or

[84] For an example, see *Elevated*, programmed by iq, for a total program size of 4 kilobytes: `https://www.youtube.com/watch?v=jB0vBmiTr6o`

presentations[85]. However, this presentation format acknowledges a kind of individual, artistic feat, rather than the *egoless programming* lauded by Brooks in professional software development[86].

Pushing the boundaries of how much can be done in how little code, here is a 256-bytes demo resulting in a minute-long music video[**?**] on the Commodore 64. It is firsted listed as a hexademical dump by its author:

```
0000000 0801 080d d3ff 329e 3232 0035 0000 4119
0000010 d01c dc00 0000 d011 0be0 3310 610e f590
0000020 0007 1fff 4114 24d5 2515 5315 6115 29d5
0000030 0f1b 13e6 13e6 02d0 20e6 61a9 1c85 20a7
0000040 3fe0 08f0 0c90 114e 6cd0 fffc 6da0 2284
0000050 d784 4b4a a81c 13a5 3029 02d0 1cc6 2fe0
0000060 11f0 02b0 02a2 10c9 09f0 298a aa03 f3b5
0000070 0a85 ab2d b000 b711 b622 9521 a500 4b13
0000080 aa0e f8cb cc86 0749 0b85 13a5 0f29 0fd0
0000090 b8a9 1447 0290 1485 0729 b5aa 85f7 a012
00000a0 b708 910d 880f f910 b7a8 9109 8803 f9d0
00000b0 7e4c 78ea 868e 8e02 d021 4420 a2e5 bdfd
00000c0 0802 0295 d0ca 8ef8 0315 cc4c a900 8d50
00000d0 d011 ad58 dc04 c3a0 1c0d 48d4 044b 30a0
00000e0 188c 71d0 e6cb 71cb 6acb 2005 58a0 d505
00000f0 cb91 dfd0 aa2b 6202 1800 2026 2412 1013
```

Even with knowledge of how hexadecimal instructions map to the instruction set of the specific chip of of the Commodore 64 (in this case, the SID 8580), the practical use of these instructions takes productive advantage of ambivalence and side-effects. In the words of the author, Linus Akesson (emphasis mine):

> We need to tell the VIC chip to look for the video matrix at address $0c00 and the font at $0000. This is done by writing $30

[85] You can find *Elevated*'s technical presentation here: `https://www.iquilezles.org/www/material/function2009/function2009.pdf`

[86] In architecture, such technical and artistic feat for its own sake, devoid of any reliable social use, is the pavillion, or the folly.

into the bank register ($d018). But this will be done from within the loop, as doing so allows us to use the value $30 for two things. *An important property of this particular bank configuration is that the system stack page becomes part of the font definition*.

Demosceners therefore tend to write beautiful, deliberate code which is hardly understandable by other programmers without explanation, and yet hand-optimized for the machine. This presents a different perspective of the relationship between aesthetics and understanding, in which aesthetics do not support and enable understanding, but rather become a proof of the mastery and skill required to input such a concise input for such an overwhelming output. This shows in an extreme way that one does need a degree of expert knowledge in order to appreciate it—in this sense, aesthetics in programming are shown to be almost often dependent on pre-existing knowledge.

6.3 Hacking and elegance

This relationship that hackers establish between the complexity of a problem and the minimization of a problem brings up the criteria of elegance. Elegance has both an abstract and a practical definition: its abstract definition relies on supposed synonyms, such as ”grace”, ”pleasantness”, ”style”, in which the focus seems to be mainly on the surface, apparently easily achieved. In terms of formal manifestations, then, elegance hints at restraint, at a minimal effort resulting in a noticeable result, for instance in the works of Balzac and Proust[72] or when referring to the design and conception of works in the high-fashion industry.

However, as we’ve seen throughout, form can provide a connection between the surface, on which it belongs and manifests itself, and the depth. From a more practical perspective, elegance is a concept that is often re-

ferred to in the so-called *hard* sciences, most notably mathematics, engineering and design. Gian-Carlo Rota, in his work on mathematical beauty, speaks to elegance in terms of how a mathematical proof is presented, and which only tangentially relates to its content[73]. Elegance is about the successive refinement of a proof after one has understood the *contents* of the proof, and is related to beauty insofar as it reveals the beauty of the proof, a beauty whose appreciation:

> requires familiarity with a mathematical theory, which is arrived at at the cost of time, effort, exercise, and *Sitzfleisch* rather than by training in beauty appreciation.

It can nonetheless manifest in at least three different ways. Elegance can be manifested in a simple notation, unusually concise or which requires minimal assumptions and computations (minimality); it can be found in an unconventional, intuitive, insightful or unifying presentation; finally, it is a manner of communicating a deep structure which outlines an approach that is highly generalizable (revelation). To these, Donald Kunth adds that elegance can only appear if the elegant program is implemented in the most suitable language, on the most suitable system[74].

These three approaches can be exemplified through the pieces of hacker code which we've touched on above. A one-liner is a simple, unusually concise notation, where the meaning and purpose of the action has been condense to serve exactly its purpose, without extraneous decorations. The example of the unexpected magic numbers and float to integer conversion in the inverse square root, as well as the ambivalent uses of byte code instructions in demoscene sources are both unexpected and unconventional at first, and yet prove to be the simplest solution to the problem at the hands of the programmer. Finally, an approach that is highly generalizable is the one taken by the creators of UNIX—everything is a text file.

While simple solutions to complex problems are usually sought after,

engineering and systems design consider an elegant solution as *the least complex, sufficient solution*[75]. Another dimension of elegance appears here: it is not just about surface presentation, nor is it about unconventional, yet intuitive approaches, but it is also dependent on sufficiency. That is, an elegant solution has for necessary condition the efficiency in the face of the problem posed. An elegant solution which doesn't solve the problem is neither elegant, nor a solution. This puts elegance back into the context of the problem domain, implying that elegance varies with contexts and requirements of a given program, written by a particular individual and individuals, in a particular context[76].

Finally, Rota provides us for a reason for which elegance is a concept that most agree exists, but few can straightforwardly define (perhaps in the same category as Alexander's Quality Without a Name), by looking at its opposite:

> Mathematicians seldom use the word "ugly." In its place are such disparaging terms as "clumsy," "awkward," "obscure," "redundant," and, inthe case of proofs, "technical," "auxiliary," and "pointless." But the most frequent expression of condemnation is the rhetorical question, *"What is this good for?"*
>
> [...]
>
> The mathematician who is baffled and asks *"What is this good for?"* is missing the sense of the statement that has been verified to be true. Verification alone does not give us a clue as to the role of a statement within the theory; it does not explain the relevance of the statement. In short, the logical truth of a statement does not enlighten us as to the sense of the statement. Enlightenment, not truth, is what the mathematician seekswhen asking, *"What is this good for?"*

Rota's *enlightenment* seems to be similar to the use of the term *understanding* we've used so far to characterize the role of aesthetics as they

are manifested throughout program texts. In hacking specifically, understanding is that of the machine, of the language, and both of the problem to be solved as well as the solution to be found. Elegant code, then, is code whose formal notation is a testament to a thorough understanding of the constraints at stake, to an enlightenment of the workings of the system; of the implications of a deep, complex structure to be skillfully dealt with, writing only what needs to be written.

---

7 Conclusion

7.1 Summary of current research

l've outlined in this document additional aspects of the relationship between the aesthetics of source code, as formal, textual manifestations, and the understanding of a program text, of which there are various kinds. To do so, I've looked at conrecte practices as well as conceptual frameworks to better qualify this relationship. In terms of practices, we've continued our overview of the different groups of people who write code, by including source code poets, hackers, and language designers—focusing slightly more on differences rather than on overlaps.

Source code poets rely mostly on metaphors in order to communicate their poetic message to the user. Narrowing down the broader field of computer-aided literary works, from interactive fiction to generative poetry, we've identified a subset of texts which focuses primarily on reading the source code itself, rather than reading its output. These texts include obfuscated code, as puzzle-like activities providing a twist on our understanding of the output, static code poetry, in which the output of the program only matters insofar as it is legal by the compiler or interpreter's rules, and active code poetry, where the output of the program, while secondary to the reading of the source, nonetheless provides additional understanding with regards to the poetics of the discourse.

To elicit such an effect, they seem to build on an extended definition of the metaphor. Particularly, we've seen the role that literary theory and contemporary definitions of metaphor can play when it comes to understanding programs, extending it from a strict literary perspective of the metaphor to a broader one. Through the lens of Lakoff's work, we've defined the metaphor as a linguistic device activating conventional conceptual structures which we all hold. With Ricoeur's work on the metaphor, we've extended it from

the strict locus of the word to that of the sentence, an extension which allows us to consider both the source and the execution of the program text as a whole metaphorical entity. In turn, the workings of these metaphorical entitities can be better understood by the concepts of double-meaning, double-coding and procedural rhetoric, providing a framework to analyze such texts and how their meaning is conveyed through *semantic compression* and dynamic processes.

In the vein of how recent literary studies have focused on the cognitive processes at play during reading, we've seen that reading program texts does not depend exclusively on the language function of the brain, even though the textual interface of those texts does provide a perspective on the connections between a surface-structure and a deep-structure. Therefore, an exclusively literary framework isn't sufficient in providing means to understand how aesthetics are manifested in the process of either understanding source code, or making source code understandable.

We then turned to another field often applied to software development: architecture. From a traditional standpoint, software architecture is concerned with the high-level organization of code as a purely functional entity, highlighting a top-down approach to structure, with little concern with implementation details. Based on this functional approach, we highlighted the limits of the *form follows function* dogma, noting that superficial beauty isn't the only way to appreciate the quality of a construction. Looking at another popular connection between architecture and software developed by Christopher Alexander, *patterns* offer another productive perspective, one which highlights *habitability* as the principal feature of a good structure, whether material or computational, and manifested through concrete, material instances.

This inquiry then led us to one of the materials of code: programming languages—the material being data and hardware. Programming language theory and research further refined our understanding of the different contexts in which source code exists, from the strict syntactical linguistic con-

struct to the broader problem domain, the work of the programmer being to reconcile both. We've also seen that some programming languages can be considered better than others, based in part on the critera of abstraction and orthogonality, and therefore provide a context and a pattern language in and of themselves. As such, they aim at providing *visual patterns of semantic significance* Changing our perspective on the topic of understanding, we've also seen that semantics in computer terms are rather implemented as secondary syntaxes, and therefore that some ways through which computers can consider code "easy to understand" is through memory and parse-tree optimizations.

It is this sort of machine-oriented beauty which hackers engage in. The practices of such a group tend to focus more on the understanding of the computer (often of a very specific physical or syntactic computer). In some cases, this results in the provision of an enabling context to their readers, shown through one-liners and large-scale open-source projects, where simplicity, clarity and conciseness are the main goals; in other cases, rather than enabling such an understanding in their readers, it focuses on the exactness and technical prowess of the code they write, with output as the proof of their precise understanding of the machine they are working with.

This led us to examine the notion of elegance in the context of source code, defined as the simplest solution given a certain problem domain. This problem domain acts as a specific context, and takes into account the goal to reach, the means to do so, and the readership of the code, whether it is writing a *writerly* text[77] which can be inhabited by other programmers, writing a poetic piece fully utilizing the expressive means of the formal symbol systems which programming languages are, or demonstrating mastery over those systems.

This leads us to sketch out the beginning of an answer to our research question: aesthetics in software are about manifesting contextual relationships between surface-structures and deep-structures, these structures being themselves conceptual (metaphors or intents) or computational, and

highly contextual.

7.2 Suggestions for next steps

From there on, I identifiy several directions for further research in order to further elaborate on this hypothesis.

First, I intend to present and analyze additional program texts, from, the different communities identified, from source code poetry to production code and "hacker code". Particularly, I would like to look at a larger codebase, such as UNIX or LaTex, to potentially identify some aesthetic features in these.

Second, the section on hackers needs to be deepened, in terms of how hackers indirectly influence other communities, in terms of how one-liners and syntactically compressed code straddle the line between understanding and obfuscation, and whether or not these are just two sides of the same cognitive coin. Additionally, I intend to examine further the parallels between architecture and hacking, particularly by looking at what architectural follies and pavillions have to tell us about the relationship of technical skill, usability and beauty.

Third, I want to conduct a deeper analysis on the aesthetics of theoretical computer science, by analyzing corpora of textbooks and academic research, by highlighting further the relationship between a theoretical understanding of computation and a hacker practice, and by more rigorously examining to what extent aesthetics might or might not matter to computer systems.

On the programming language side, I see three additional directions. I would map out more clearly the framework provided by Goodman's *Languages of Art* onto the semantic affordances of programming languages. Building on elegance as a tool for mathematical and scientific understanding, I would further highlight the role of style in programming, specifically from Gilles-Gaston Granger's perspective. Finally, I would also further in-

vestigate to what extent programming languages themselves can be considered textual/semantic patterns in themselves, or if they only afford semantic patterns in source code, or both.

References

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

[2] Will Crowthers. *Colossal Cave Adventure*. 1977.

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

[4] Olga Goriunova and Alexei Shulgin. *Read Me: Software Art & Cultures*. Aarhus University Press, Aarhus, 2004th edition edition, December 2005.

[5] VILÉM FLUSSER and Rodrigo Maltez Novaes. *On Doubt*. University of Minnesota Press, 2014.

[6] Sharon Hopkins. Camels and Needles: Computer Poetry Meets the Perl Programming Language. *Usenix Technical Conference*, 1992.

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

[8] Philippe Bootz. *The Problem of Form Transitoire Observable, A Laboratory For Emergent Programmed Art*. 2005.

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

[10] I. Bogost. *The Rhetoric of Video Games*. 2007.

[11] Jeremy Tirrell. Dumb People, Smart Objects: The Sims and The Distributed Self. In *The Philosophy of Computer Games Conference*, 2012.

[12] George Lakoff. *Metaphors We Live By*. University of Chicago Press, 1980.

[13] Paul Ricoeur. *The Rule of Metaphor: The Creation of Meaning in Language*. Psychology Press, 2003.

[14] James Jerome Gibson. *The Ecological Approach to Visual Perception*. Psychology Press, 1986.

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

[16] Michael Polanyi and Amartya Sen. *The Tacit Dimension*. University of Chicago Press, Chicago ; London, revised ed. edition edition, May 2009.

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

[18] Paul du Gay, Stuart Hall, Linda Janes, Anders Koed Madsen, Hugh Mackay, and Keith Negus. *Doing Cultural Studies: The Story of the Sony Walkman*. SAGE Publications Ltd, Los Angeles, CA, second edition edition, June 2013.

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

[20] Terry Winograd. *Language As a Cognitive Process: Syntax*. Addison-Wesley, Reading, Mass, May 1982.

[21] Walter Kintsch and Teun A. van Dijk. Toward a model of text comprehension and production. *Psychological Review*, 85(5):363–394, 1978.

[22] Noam Chomsky. *Aspects of the theory of syntax*. Cambridge, M.I.T. Press, 1965.

[23] Anna A Ivanova, Shashank Srikant, Yotaro Sueoka, Hope H Kean, Riva Dhamala, Una-May O'Reilly, Marina U Bers, and Evelina Fedorenko. Comprehension of computer code relies primarily on domain-general executive brain regions. *eLife*, 9:e58906, December 2020.

[24] Nelson Goodman, , and Others. Basic Abilities Required for Understanding and Creation in the Arts. Final Report. September 1972.

[25] Kathleen Forsythe. Cathedrals in the Mind: The Architecture of Metaphor in Understanding Learning. In Robert Trappl, editor, *Cybernetics and Systems '86: Proceedings of the Eighth European Meeting on Cybernetics and Systems Research, organized by the Austrian Society for Cybernetic Studies, held at the University of Vienna, Austria, 1–4 April 1986*, pages 285–292. Springer Netherlands, Dordrecht, 1986.

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

[27] David Garland. Software Architecture: A Roadmap. In *The Future of Software Engineering*. ACM Press, anthony finkelstein (ed.) edition, 2000.

[28] Friedrich A. Kittler. There Is No Software. In *Literature, Media, Information Systems: Essays*, pages 147–155. Amsterdam Overseas Publishers Association, Amsterdam, john johnston edition, 1997.

[29] Diomidis Spinellis and Giorgios Gousillos. *Beautiful Architecture*. O'Reilly Media, 2009.

[30] Eric S. Raymond. *The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. "O'Reilly Media, Inc.", 2001.

[31] Frederick Phillips Brooks and Frederick P. Brooks Jr. *The Mythical Man-month: Essays on Software Engineering*. Addison-Wesley Publishing Company, 1975.

[32] Harold Osborne. The Aesthetic Concept of Craftsmanship. *British Journal of Aesthetics*, 17(2):138, 1977.

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

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

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

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

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

[38] John McCarthy, Michael I. Levin, Paul W. Abrahams, Massachusetts Institute of Technology Computation Center, and Daniel J. Edwards. *LISP 1.5 Programmer's Manual*. MIT Press, 1965.

[39] Louis H. Sullivan. *The tall office building artistically considered*. 1896.

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

[41] Peter Downton. On Knowledge In Architecture and Science. 1998.

[42] Christopher Alexander. *The Timeless Way of Building*. Oxford University Press, 1979.

[43] Christopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, New York, January 1977.

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

[45] Paul Taylor. Patterns as Software Design Canon. *ACIS 2001 Proceedings*, January 2001.

[46] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, and Grady Booch. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, Reading, Mass, 1st edition edition, November 1994.

[47] Gilles-Gaston Granger. *Essai d'une philosophie du style*. Odile Jacob / Seuil, Paris, édition revue et corrigée edition, January 1988.

[48] Peter Van Roy. Programming Paradigms for Dummies: What Every Programmer Should Know. 2012.

[49] Terrence W. Pratt and Marvin V. Zelkowitz. *Programming Languages: Design and Implementation*. Pearson, Upper Saddle River, NJ, 4th edition edition, September 2000.

[50] Robert W. Sebesta. *Concepts of Programming Languages*. Pearson, NY, NY, 12th edition edition, February 2018.

[51] Niklaus Wirth. The Essence of Programming Languages. In László Böszörményi and Peter Schojer, editors, *Modular Programming Languages*, Lecture Notes in Computer Science, pages 1–11, Berlin, Heidelberg, 2003. Springer.

[52] Ryan Stansifer. *Study of Programming Languages, The*. Prentice Hall, Englewood Cliffs, N.J, 1st edition edition, July 1994.

[53] Evelina Fedorenko, Anna Ivanova, Riva Dhamala, and Marina Umaschi Bers. The Language of Programming: A Cognitive Perspective. *Trends in Cognitive Sciences*, 23(7):525–528, July 2019.

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

[55] Bruce J. McLennan. "Who Care About Elegance?": The Role of Aesthetics in Programming Language Design. Technical Report UT-CS-97-344, University of Tennessee.

[56] V. N. Voloshinov and Michail M. Bachtin. *Marxism and the Philosophy of Language*. Harvard University Press, 1986. Google-Books-ID: fIPuRyFvDKIC.

[57] Martin Sustrik. 250bpm, 2021.

[58] Ravi Sethi. *Programming Languages: Concepts and Constructs*. Reading, Mass, 1996.

[59] Peter Suber. What is Software? *Journal of Speculative Philosophy*, 2(2):89–119, 1988. Publisher: Pennsylvania State University Press.

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

[61] James H. Moor. Three Myths of Computer Science. *British Journal for the Philosophy of Science*, 29(3):213–222, 1978. Publisher: Taylor & Francis.

[62] Edward A. Lee. The Problem with Threads. Technical Report UCB/EECS-2006-1, EECS Department, University of California, Berkeley, January 2006.

[63] Steven Levy. *Hackers: Heroes of the Computer Revolution - 25th Anniversary Edition*. "O'Reilly Media, Inc.", May 2010.

[64] Ron Rosenbaum. Secrets of the Little Blue Box | Esquire | OCTOBER 1971.

[65] Pekka Himanen. *The Hacker Ethic And The Spirit Of The Information Age*. January 2001.

[66] Joseph Weizenbaum. *Computer Power and Human Reason: From Judgment to Calculation*. W H Freeman & Co, San Francisco, 1st edition edition, March 1976.

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

[68] Eric Raymond. *The Art of UNIX Programming*. Addison-Wesley, Boston, 1st edition edition, September 2003.

[69] Peter Seibel. *Coders at Work: Reflections on the Craft of Programming*. Apress, September 2009.

[70] John Lions. *Lions' Commentary on UNIX 6th Edition with Source Code*. Peer-to-Peer Communications, 1996.

[71] Markku Reunanen. *Computer Demos - What Makes Them Tick?; Tietokonedemot - mikä saa ne hyrräämään?* G3 Lisensiaatintyö, Aaltoyliopisto; Aalto University, 2010.

[72] Juliette de Dieuleveult. Les artistes de l'élégance chez Marcel Proust et Vladimir Nabokov. *Revue de littérature comparée*, n o 303(3):301–321, 2002. Place: Paris Publisher: Klincksieck.

[73] Gian-Carlo Rota. The Phenomenology of Mathematical Beauty. *Synthese*, 111(2):171–182, 1997. Publisher: Springer.

[74] Donald E. Knuth. *Literate Programming*. Center for the Study of Language and Inf, Stanford, Calif., 1st edition edition, June 1992.

[75] M. Efatmaneshnik and M. J. Ryan. On the Definitions of Sufficiency and Elegance in Systems Design. *IEEE Systems Journal*, 13(3):2077–2088, September 2019. Conference Name: IEEE Systems Journal.

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

[77] Roland Barthes. *S-Z*. Hill & Wang, New York, May 1977.