Thesis check-in - Role of aesthetics in understanding source code

Pierre Depaz

March 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 (INCLUDE FINDINGS IN THREE.MD)—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, 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 Source code poetry

This section focuses on source code poetry, as the closest use of ”literary arts” in code. We will see how this particular way of writing software, to an explicitly aesthetic end, rather than a functional one, summons particular claims to art and beauty. These claims maintain a complex relationship to the nature and purpose of code, 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 and further definition of the field, I will highlight to particular source code poems, chosen for their meaning-making affordances, and conclude on the aesthetic standards at play in their reading and writing.

2.1 Overview

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 In line with the framework of this research, the focus we propose here is shifted from output to input, for executable binary to latent source.

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 acts of writing 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 focused more on the playable output of the software, rather than on its source code. Written in FORTRAN 4 exhibits several features which wouldn't fit within our typology previously established, particularly in terms of variable naming (e.g. variables named 'KKKT', 'JSPK', or 'GOTO' statements, since then considered harmful[1]). Its source code was indeed only examined due to the recognition of the cultural influence of the game, decades later. Similarly, 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 and recognized by the interactive fiction community, nonetheless consists in a single HTML document, comprising well-formatted and understandable HTML and CSS markups, and three single lines of "uglified" JavaScript[2]. In the case of visual poetry, one can see how the source code of works such as bpNichol[3], is dictated exclusively by the desired output, with a by-product of visually pleasing artifacts throughout the code as foreshadowing the result to come[4]. These examples of in-

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

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

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

[4] another artefact is the subroutine at line 1600, an "offscreen romance" only visible in the source.

teractive fiction, 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 closer 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.

These works and their authors build on a longer tradition of rule-based composition, from Hebrew to the Oulipo and John Cage, amongst many others[3], a tradition in which creativity and beauty can emerge from the strict application of 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. The poem exists through the code, but the poem still isn't exclusively limited to the humanly-readable version of the code, as it only comes to life once interpreted or compiled by a machine. 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[5]. These do constitute a body of work centered around the concept of generative aesthetics[4], in which beauty comes from the unpredictable

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

and somewhat complex interplay of rule-based systems, and 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 more pure surface version of it (both code poetry slam and obfuscated c code)
- address esolangs
- then focus on the ones that i find the most relevant (SAY WHY) (code poems, perl poetry, speaking code[5])

2.2 Analysis of a poem

pick the most popular one (sharon hopkins? who wrote one of the first perl poems)

also include a list of corpora

and then pick one i like in the speaking code

and then pick one i like in the code poems book

2.3 Aesthetic standards in source code poetry

particular, *double-coding* and *double-meaning*.

3 Metaphors

3.1 In literature and software

Essentially a lot of Lakoff, and then the advance into cognition and literature. I also need a source on traditional metaphors.

https://issuu.com/instituteofnetworkcultures/docs/tod14-binnenwerk-def-pdf

3.2 The place of imagination

Expand upon it both in lit and in code. How do they differ? Ambiguity

3.3 Mental models

What is a mental model? What is knowledge?

Understanding: *deep structure* vs. *surface structure*

And then transition into metaphors as architectures of thought.

4 Architecture

This section is mostly based on the work of *patterns*, both in softdev and in architecture, and how they can be considered both functional and beautiful; furthermore I will develop on how these two relate (through habitability, QWaN).

4.1 General software architecture

Emergence of the field, planning vs. non planning.

Planning as a consequence of the structured revolution.

Non-planning as a consequence of low-barrier to contribution.

4.2 Beauty in architecture

Highlight how uncertain, elusive it is. And then focus a big part on Alexander's work.

Movement of people is also some kind of structure

4.3 Applying architectural beauty in software

A lot of Gabriel's work, his connection to poetry, and also looking at how it applies well (softdev), and not so well (code poetry → because the func-

tionality of code poems isn't quite so obvious as the functional use of a building, or of a program)

Transition with the case of the case of hacking: code that is not meant to be read.

5 Hacking

Is there beauty in inscrutability? Particularly, this redirects to the understanding of the machine (e.g. trying to reduce character counts for one-liners).

5.1 History of hacking

And a disambiguation of the term.

5.2 Unreadability and aesthetics

This shows you need a degree of expert knowledge in order to appreciate it.

Example of the one-liners.

Example of the demoscene.

5.3 Hacking and/or the lack of beauty

Why is there ne beauty in hacking? Does there need to be beauty anywhere?

There doesn't seem to be so much because it's so much focused on what the machine understands; the real beauty is how they make the humans *realize* what the machine *really* understands.

6 Machine understandings

This is about whether or not the machine really understands anything.

UNIX → you are not expected to understand this

6.1 Computation

What is a computer? what is computation? There are differring approaches, and I should highlight what is mine (that of the formal, symbolic system). Cantell-Smith can be good here, if alone to explain that things are complicated to explain, but nonetheless intuitive to any serious practitioner of CS.

6.2 Programming languages

First, by what makes a PL (a bit of history, a bit of theory)

Second, what makes a good PL.

6.3 Formal systems

Start by what is a formal system, then show two consequences: first, it fits with goodman and second, it actually seems to have trouble with meaning.

7 Programming Semantics

Answer the question as to whether they have semantics (they do according to the specs, but it's just another *décalage*)

7.1 Semantics in PL

Approach it first in a very practical way (parse trees, tokens, environments), and then in a more theoretical way (what are we even trying to communi-

cate?)

7.2 Concepts of PL

The issue actually comes from the fact that some concepts might be very foreign and hard to communicate (which is why programming can be hard, and how the whole section has shown relative limitations of doing so).

And so this is why we need aesthetics: to communicate both easy and complicated things, to reduce friction as much as possible. Example: Alan Perlis's Epigrams are poem-like sentences.

Another example if thread concurrency. The book on parallel programming mentions an example of parallel thread processing which looks beautiful but is ugly on the inside: https://mirrors.edge.kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook-e2.pdf p. 105

8 Conclusion

Sum up a bit all that has been said.

aesthetics matter, even in such a highly formal, syntactical, autotelic system as a computer. they connect a surface-structure with a deep structure.

how does it contribute to the world? by showing that there is no separate domain of aesthetics, but also that they're not essential, but a mark of high-quality and, again *that they allow us to understand*.

Recap on some of the concepts:

- semantic compression - spatio-visual problem solving

Finishing on the MIT study and Goodman:

> The emphasis placed on the symbolic, cognitive, planning aspects of the arts leads us to give value to the role played by

problem-solving, seeing there a model in terms of which the moment-to-moment artist's behavior at work can be described. "An analysis of behavior as a sequence of problem- solving and planning activities seems to be most promising [...]" (goodman)

8.1 Next steps

Close-reading of more source code.

Gathering of more hacking resources and computer science/abstract resources.

References

[1] Francoise Detienne. *Software Design – Cognitive Aspect*. Springer Science & Business Media, December 2012. Google-Books-ID: xh3vBwAAQBAJ.

[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] Geoff Cox and Christopher Alex McLean. *Speaking Code: Coding as Aesthetic and Political Expression*. MIT Press, 2013. Google-Books-ID: wgnSUL0zh5gC.