The Aesthetics of Source Code - Pierre Depaz

Further definition of topic and future outline of work - 08.2020

Aesthetics and understanding

At the beginnning of the Spring semester 2020, I had established a clear research direction, directed at what role aesthetics have in the process of understanding source code. While the definition of aesthetics upon which this research relies is based on *aesthetics as a physical manifestation which can be grasped by the senses*. The limitation of this starting point is justified mainly by the object of this study: as an object, rather than as a concept. By approaching source code as an object (specifically, as multiplicity of objects, “texts” written and read), rather than as a concept, I therefore put its immediately graspable aspects in the foreground. While the social, cultural, intellectual and emotional components are still significant in the appreciation of beauty in source code, the comparative lack of close examination of *how* code is written is the justification for such a definition of aesthetics. In the following, the adjective “beautiful” when qualifying code will mean “code which presents aesthetic qualities”.

What still needed to be defined, however, are two things: a first set of criteria of how “beautiful” source code should look like, and their relationship to “understanding”. The work conducted this semester has therefore focused on the gathering and examination of the corpus of source code texts, along with their accompanying explanations, justifications and overall meta-texts, in order to find out how are references to “beauty” and “understanding” made. Amongst the vast majority of the corpus elements, practicioners tend to present or discuss a piece of source code which they consider “beautiful”, “aesthetically pleasing”, and accompany this presentation with justifications about *how* to make a piece of code beautiful and/or *why* make a piece of code beautiful. It is these discourses that are used to elaborate on what source code aesthetics look like *in practice*, as well as what kind of role they play in the life of source code text.

This process has led to the constitution of an initial set of aesthetic properties that are repeatedly highlighted by a certain sub-set of practictioners. In effect, the group of those who write and read source code is far from being homogeneous, and can actually be grouped into distinct categories with distinct practices and standards[1]. While additional sources establish their own distinctions[2][3][4], the multiplicity of contexts within which code is written leaves litte doubt. Leaving aside for now a thorough defintion of each of these, I've identified four main categories of individuals writing and reading source code, which I group under the umbrella term *code practicioners*. These categories include: software engineer, academic, hacker and artist. These categories intend to provide heuristics, rather than strict definitions, and each of these categories can overlap within one individual or group of individuals.

The sub-sets of practicioners examined so far include software engineers and artists—and thanks to the aforementioned overlap of categories, I am making the hypothesis that the initial findings made through the comparison of how aesthetics are conceived of by engineers and artists will be reinforced and further qualified during the examination of how the remaining categories (academics and hackers)[5].

[1] https://www.americanscientist.org/article/cultures-of-code

[2] https://josephg.com/blog/3-tribes/

[3] https://blog.codinghorror.com/the-two-types-of-programmers/

[4] https://mkdev.me/en/posts/the-three-types-of-programmers

[5] these categories match what Wittgenstein has called *forms of life*: socio-cultural contexts of use, underpinned by normative activities (e.g. technical writing)

Established software engineering practices

The vast majority of code written today has been done by software engineers. While not the only group of people to write and read code, they are by far the most significant. The appearance of the profession in the late 1950s and early 1960s, emerging from a purely academic or military activity, brought with it a change in discourses relating to how code should be written, most eloquently by E. W. Djikstra [6], along with Knuth [7], Kernighan [8] and Martin [9] amongst others. Since these earlier texts focused on defining the practice of software development at a professional level, the inclusion from the get-go of an artistic component (*“The Art of Programming”*), as well as a cognitive one (*“GOTO Statement Considered Harmful”*) is a significant indicator that writing software isn’t an exclusively mechanical activity. However, while claiming in its introduction that writing code is an art, *The Art of Programming* doesn’t address *what* exactly it is that makes writing code an artistic process and source code a beautiful object, and therefore leaves room for the following investigation. Following Knuth’s claim, this question of “beautiful code” has been addressed by members of the profession itself, at various levels: a couple of monographs[10][11], conferences, academic articles, blog posts and Q&A websites.

[6] Dijkstra, Edsger W., Chapter I: Notes on structured programming. Structured programming. Academic Press Ltd., GBR, 1–82. 1972.

[7] Knuth, Donald, The Art of Programming, Vol. 1, Addison-Wesley, 2001.

[8] Kernighan, Brian W. and Plauger, P. J., Elements of Programming Style, McGraw-Hill, 1978.

[9] Martin, Robert C., Clean Code, Pearson, 2008.

[10] Oram, Andy and Wilson, Greg (ed.), Beautiful Code: Leading Programmers Explain How They Think, O’Reilly Media, 2007.

[11] Chandra, Vikram, Geek Sublime, Graywolf Press, 2014.

The analysis of this corpus has led to mulitple insights for this research project: qualifying the purpose of aesthetics for software practicioners, establishing an aesthetic framework and providing further insight into the relationship between aesthetics and understanding.

First, it has helped anchor further what role aesthetics play for one of the categories of software practitioners. These findings complement the statements by more leading figures among software development, and confirm, or qualify the statements of said figures. Particularly, it has highlighted that, while aesthetics are important in any source code, they are often difficult to achieve and are always secondary to the functionality of the program (i.e. does it actually work?). Any presence of aesthetic features should not be there for their own sake, but rather contribute to a particular goal: that of facilitating the communication of the writer’s mental model of the problem.

Second, it has allowed me to constitute a significant part a set of features of beautiful code. When offering their opinion on the oft-repeated topic of beautiful code, the comments and explanations of code in the corpus do not contain uni-dimensional criteria, but rather criteria which can be applied at multiple levels of reading. Some of those tend to relate more to the over-arching design of the code examined while others, closer to our working definition of aesthetics, focus on the specific formal features exhibited by a piece of source code. This variety of criteria led me to base the framework of aesthetic criteria on John Cayley’s distinction between *structures*, *syntaxes* and *vocabularies*[12]. Cayley’s framework allows me to take into account an essential aspect of code: that of scales at which aesthetic judgment operates. Additionally, it also provides a bridge with literature and literary studies without imposing too rigid of a grid. While it does not immediately acknowledge more traditional literary concepts such as fiction,

[12] Cayley, John, The Code Is Not The Text (Unless It Is The Text), Electronic Book Review (ebr), 2002.

authorship, literarity, etc., it does leave room for these concepts to be taken into account. Particularly, we'll see that the concept of authorship (who writes to whom) will be useful in the future.

Third, the analysis has refined the relationship between "aesthetics" and "understanding". The necessity for code to be understood, the desire for code to be beautiful and the desire for code to be functional are often intertwined and highlight how the first hints at the realization of both clarity and functionality. This idea that "something which is beautiful is something which works" is a central one in Nelson Goodman's work[13], and this work serves as a theoretical backbone for the analysis of our object of research. However, even though this relationship between beauty, clarity and functionality seems to be an argument in support of aesthetics as a means of efficiently communicating concepts, it will become clear that, in the case of source code, that there are multiple (and sometimes conflicting) aesthetic criteria across software practicioners. The hypothesis here is that it is due to the fact that the concepts that are being communicated are themselves plural in nature (immediate function, theoretical illustration, unique skill and artistic cosmogony).

In the next section, I move to further explain the perceived relationship between beauty and code among software practicioners.

The role of beauty in code

"Aesthetics alleviate cognitive pain"[14]. This excerpt from *The Art of Readable Code*, presented as an industry manual for professional software developers, sums up the overall sentiment of software developers as surveyed through my corpus analysis. While most code can be understood,

[13] Goodman, Nelson, Languages of Art, Hackett Publishing Company, Inc., 1976.

[14] Boswell, Dustin, The Art of Readable Code: Simple and Practical Techniques for Writing Better Code, O'Reilly Media, 2011.

sometimes after considerable effort, beautiful code supposedly bypasses any need for additional explanation, reaching a highly sought-after status of “self-explanatory”. This status for aesthetics to ease understanding of the text also answers one of the early research questions of this thesis regarding the necessity for code to be beautiful in the first place. While the existence of beautiful code quickly manifested itself at the beginning of this research, two questions then seemed to arise due to its close connection to complexity, intelligibility and understandability.

First, *what and how do aesthetics in code make intelligible*? What is made intelligible isn’t exclusively what the program *does*, but can also refer to the knowledge of an existing algorithm, a given idiomacy in a programming language, an architecture of hardware, a practice of reading and writing of fellow programmers or a certain conception of the world. That is, what should be made intelligible is both an intent and a mental model, within a particular socio-technical context composed at least of a writer, a reader, a language and a hardware. How it makes such an action understandable is addressed in the following section, in which I sketch out a typology of aesthetic criteria, before highlighting how some of these criteria can denote different intents and mental models.

Second, *do aesthetics in any kind of code always aim at making intelligible*? That is, is the aesthetic in the code exclusively transitive, relating to something other than itself, or intransitive, referring only to itself? This opens up a further discussion on whether or not functionality is an essential part of aesthetics (i.e. *“beauty that you can use”*[15]), and to what extent there are contexts and mediums in which beauty can exist without an external aim.

[15] Oram, Andy and Wilson, Greg, op. cit.

There thus seems to be a necessity for beauty: since programming is an inherently complex activity[16], dealing with abstract concepts, and dealing with them as raw materials, an aesthetically pleasant piece of source code is therefore an easily-understandable one, enabling the development of mental models[17] in the reader, mainly through the process of visual, syntactical and semantic metaphors, connecting immediate sensual manifestations to abstract, shared ideas[18]. These metaphors arise from particular constraints which writers of code face. The issue here is to communicate what the program is (as a conjunction of *what* a program does, *how* a program does it and *why* it does so). This essence of the code is based on the writer-programmer's mental model of the problem at hand, and aimed at the reader-programmer, and this communication happens a highly-restricted symbolic language; theoretically able to compute any finite problem, while at the same time limited in practice to instruction sets and syntax which doesn't adequately cover the need for expressing the intent of the work.

Source code thus needs to communicate something beyond itself. This can be what the code does, how and why it does it, and how it approaches the problem domain. The object, the manner and the context are what the reader focuses on, and not aesthetics features in themselves; that is, if the code were to be written differently, it would not ultimately harm the purpose of reading it (even though it would still make the process of doing so more cumbersome).

However, one of the reasons for which a code-text exists and which relies first and foremost on aesthetics is the skill of a given writer. Just because a significant part of code can be considered a semantic interface which

[16] Dijkstra, Edsger W., Craftsman or Scientist? retrieved from University of Texas

[17] Forrester, Jay Wright, World Dynamics, Wright-Allen Press, 1971.

[18] Goodman, op. cit.

should ultimately become as invisible as possible[19] (self-explanatory, it allows the reader to get directly to the problem, without stumbling on the syntax), the aesthetic nature of a code-text is considered by most practicioners as a testament to the skill of a writer.

Before we move on to a more detailed analysis of what makes source code aesthetically pleasing in the following section, I would like to point out a recurring reference made by practicioners regarding the general heuristic of writing beautiful code. **Elegance**, always loosely defined, is seen as doing the least with the most; as the number of lines of code diminish, each of them become more and more essential[20]. Throughout the corpus analysis, various references to Antoine de Saint-Éxupéry's quote appear regularly: *"Perfection is achieved, not when there is nothing more to add, but when there is nothing left to take away."*. Beyond highlighting a desire for a literary connection, this citation also hints at the iterative process of writing code: adding, then removing.

Such a process is echoed in the practices of craftsmanship, rather than scientific approach. Indeed, several[21] authors[22] have alluded at programming as a craft[23], but haven't focused specifically on the parallels in form. It thus seems necessary not to oversee that connection, and to ask *what are the relationships between the aesthetic criteria of craftsmanship the aesthetic criteria of source code?*. Some of these criteria include clarity, mastery, cooperativity and utility and constitute an additional axis of research. An early paradox in this relationship concerns the aim of the code itself, the criteria of mastery excludes, for some, any code-text which

[19] Galloway, Alexander, The Interface Effect, Polity, 2012.

[20] Nabokov, Vladimir, Lectures on Literature, Mariner Books, 2002.

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

[22] Dijkstra, craftsman vs. scientist

[23] Software Craftsmanship

doesn’t solve an interesting problem (e.g. *“a user-login form cannot be beautiful”*). Craftsmanship would allow for mundane uses to be the vehicle of aesthetic manifestation (e.g. a chair, an illustration), but the field of computation within which code evolves gives the possibility of always pointing towards bigger, newer computing problems which can be addressed. In terms of resemblance, though, both craftsmen and software developers abide by deeply-engrained, bottom-down mottos and heuristics such as DRY (*Do Not Repeat Yourself*), KISS (*Keep It Simple, Stupid*) and SOLID[24], which act as essential guideline for writing software in a professional context and underpin, as we will see, all other aesthetic standards in this community of practicioners.

Aesthetic criteria

As mentioned above, the corpus of textbooks, essays, online blog posts and comments addressing what makes source code beautiful or aesthetically pleasing has been analyzed according to three main categories: structure, syntax and vocabulary, as all related to formal manifestations in source code, both functioning at different scales. *Structure* is defined by the relative location of a particular statement within the broader context of the code-text, as well as the groupings of particular statements in relation to each other and in relation to the other groups of statements within the code-text. This also includes questions of formatting, indenting and linting as purely pattern-based formal arrangements. *Syntax* is defined by the local arrangement of tokens within a statement, including control-flow statements (and therefore not restricted to single-line statements). It also includes language-specific choices, referred to as idioms, and generally the type of statements needed to best represent the task required (e.g. using an `Array` or a `struct` as a particular data structure for a collection of things). Finally, the *vocabulary* refers to the user-defined elements of the source

[24] single responsibility of classes, open/close principle, liskov substitution, interface segregation, dependency inversion

code, in the form of variables, functions, classes and interfaces. Unlike the two precedent categories, this is the only where the writer can come up with new tokens, and is the closest to metaphors in traditional literature.

- Structure

Structure, as the highest-level group of criteria, is both easy to grasp and somewhat superficial: most of the criteria which compose it are *indicators* and not *proof* of beautiful code, indeed necessary, but not sufficient. Structure itself can be further separated between surface-structure, and deep-structure. The criteria for beauty in surface-structure is **layout**, as the spatial organization of statements, through the use of line breaks and indentations. While serving additional ends towards understanding, proper layout (whether according to conventions, or deliberately positioning themselves against these conventions) seems to be the first requirement for beautiful code. In terms of aiding understanding, blank space creates semantic groupings which enable the reader to grasp, at a glance, what are the decisive moments[25] in the code's execution, and presented by some as akin to paragraphs in litterature[26]. Any cursory reading of source code always first and foremost judges layout.

This aid to understanding is further highlighted by a deep-structure criteria of **conceptual distancing**: statements that have to do with each other are located close to each other. As such, visual appearance reflects the conceptual structure of the code (and, indeed, some argue that the data which the code processes predates the code itself in dictating its layout). While an over-arching principle, it is vague enough to be open to interpretation by practitioners and is therefore unable to act as a strict normative criteria (e.g. should every code-text follow the *stepdown rule of function declaration* or *alphabetical rule of function declaration* when

[25] Sennett, Richard, op. cit.

[26] Matsumoto, Yukihiro, Treating Code As An Essay, in Beautiful Code, op. cit.

writing in a language which doesn’t enforce it? should local variables all be declared at the beginning of the highest scope at which they belong, or at the closest location of their next use? should all data be prepared, and then processed, or should each data be prepared and processed in each of their contexts?).

The related criteria of *local coherence* (what is next to each other is related to each other) echoes similar questions about the coherence and consistence in Goodman’s aesthetic theory[27]. Local coherence enables what Goodman calls semantic density, in which tokens grouped together obtain a greater denotative power. Local coherence seems at first to stand at odds with the undesirable but unavoidable *entanglement* of code. Indeed, proponents of local coherence in source code imply that a beautiful piece of code should not have to rely on input and output (i.e. not be entangled) and therefore be entirely autotelic. Such an assumption runs contrary to the reality of software development as a practice, and as an object embedded in the world, and thus not “usable” by software developers. Such an isolated approach to code doesn’t match the claims above that code should be useful, and so is probably intended to be understood in a more flexible manner (see in vocabulary below: function arguments).

A correlate to conceptual distancing is **conceptual symmetry**, which states that groups of statement which do the same thing should look the same. It then becomes possible to catch a glimpse of patterns, in which readers know what does what according to a brief overview. Conceptual distancing can be further improved by **conceptual uniqueness (unicity?)**, which demands that all the statements that are grouped together only refer to one single action: complex enough to be useful, and simple enough to be graspable (possibly the start of a definition of *elegance*). Following this, then, beautiful code is *“the code that does the job while using the least amount of different ideas”*, which, according to the DRY principle, implies a

[27] Goodman, Nelson, op. cit.

linear relationship between the number of lines of code and the amount to be understood. This is possibly an aesthetic standard, but it is unclear to what extent it is a sublime standard.

Interestingly, this last statement contradicts another aesthetic that exists among different software practicioners—hackers. In their case, beautiful code is the code which manages to pack the maximum number of ideas in a minimal amount of lines of code, both in obfuscation practices and in the writing of one-liners. This apparent conflict between clarity and complexity can be resolved in certain codebases, in which the lines of code are few, but the implications are many[28].

- Syntax

Syntax, as the mid-level group of criteria, deals most specifically with the two main components of the implementation: the algorithm and the programming language. Beautiful syntax seems to denote a conceptual understanding of the tools at hands to solve a particular problem (Knuth states that these understandings are the ones that make writing code an art, and has devoted his magnum opus to the study and communication of algorithms). Both algorithms and languages are seen as tools since, again, it is through its use and implementation that a piece of code is considered functional, and can thus be examined for aesthetic purposes. Due to this need for implementation, I will argue that algorithms exist independently from languages, but that their aesthetic value in the context of this research cannot be separated from the way they are written, and the language they are written in. Indeed, most algorithms are expressed first as pseudo-code and then implemented in the language that is most suited to a variety of factors (speed, familiarity of the author, suitability of the syntax); this seems to be a contemporary version of the 1950s, when computer scientists would

[28] for instance, `forkbomb.pl`, or the lisp interpreter

devise those algorithms through pencil and paper, and then leave their implementation at the hands of entirely different individuals—computers.

Beautiful syntax in code responds to this limitation. Since algorithms must be implemented in a certain context, with a certain language, it is the task of the writer to best do so with respect to the language that she is currently working in. In this case, knowledge of the language-as-tool and its relevant application makes beautiful syntax an *idiomatic syntax*. This involves knowing the possibilities that a given language offers and, in the spirit of the craftsmanship ethos noted previously, working with the language rather than against it. These sets of aesthetic criteria thus become entirely dependent on the syntactical context of the language itself, and can only be established with regards to each languages (e.g. knowing which keywords shouldn't be used, such as `unless` in perl, or `*` in C, knowing when to use decorators in python, the spread `...` operator in ECMAScript, etc.). Here, syntax also follows the idea of conciseness that has been touched upon at the structure-level: a writer can only be concise if she knows how the language enables her to be concise, and knowing the algorithm and the problem domain will not help to match this criteria. To what extent a syntax is **idiomatic** syntax is therefore a good indicator of the aesthetic value of a code-text, while refraining from being too idiomatic (often referred to as "clever code" and generally frowned upon).

It is difficult to establish a hierarchy between separate idioms, since they operate under different paradigms and assumptions. A developer who finds that she can best communicate her ideas according to Java will find Java beautiful. A developer who finds that she can best communicate her ideas while writing in Go will find Go beautiful. This state of affairs seems to be part of the reason as to why online platforms are full of "which language is better?" endless discussions. A syntactical criteria which acts as a response to these discusssions is **consistency**. While there might be specific reasons as to why one would want to be writing code one way or another (e.g. calling functions *on* objects rather than calling functions *from* objects in order to prevent output arguments), this minor increase in aesthetic value—through display of skill and personal knowledge— doesn't compensate for the possible increase in cognitive noise if those different ways of writing are used alternatively in an arbitrary manner. In this context, consistency prevails over efficacy, and nonetheless hints at the fact that aesthetics in source code in this context is a game of tradeoffs.

Beyond the state of syntactic consistency, the question of **linguistic reference**, bringing heuristics from one language to another is yet another aesthetic criteria. Being able to implicitly reference another language in a code-text (e.g. *"this is how we do it now that we have C++, but the current code is written in C, so one can bring in ideas and syntax that are native to C++"* or *"since Ruby can qualify as a Lisp-like language, one can write lambda functions in an otherwise object-oriented language"*), a code-switching of sorts, can both communicate a deep understanding of not just a language, but an ecosystem of languages while satisfying the purpose of maintaining clarity, assuming a certain skill level in the reader. This communicates a feeling of higher-understanding, akin to perceiving all programming languages as ultimately just "tools for the job" and whose purpose is always to get a concept across minds as fully and clearly as possible. However, a misguided intention of switching between two languages, or a mis-handled implementation can propel a code-text further down the gradient of ugliness. The concept communicated would in such a case be obscured by the conflicting idioms (e.g. writing p5.js—as a JavaScript implementation of a Java-based syntax—within an HTML document forces the co-existence of two distinct syntaxes which are made to cohabit more for purposes of platform-distribution rather than code clarity), reveal of lack of mastery of the unique aspects of the working language(s), and therefore fail to fulfill the aesthetic criterion of idiomaticity.

Finally, a syntax with high aesthetic value is a syntax which favors **natural language reading flow**. For instance, of the two alternatives in Ruby: `if people.include? person` vs. `if person.in? people`, the second one is going to be considered more beautiful than the first one, since it adapts to the reader's habit of reading human languages. However, the essential succintness and clarity of source code is not to be sacrificed for the sake of human-like reading qualities, such as when writers tend to be overly explicit in their writing. Indeed, a criteria for ugliness in code-text is *verbosity*, or useless addition of statements without equivalent addition of functionality. This testifies to the need for a **balance between machine idioms over human idioms**, with the sweet spot seemingly being the point at which machine idioms are presented as human-readable.

- Vocabulary

Vocabulary, as the only component in this framework which involves words that can be (almost) entirely invented by the writers, is often the most looked at in the literature regarding beautiful code among professional software developers. Aesthetics here deal mostly with beautiful names, and respect for conventional knowledge. It is the level of aesthetic standards which takes into account first and foremost the readership of a given code-text.

Of the two big problems of programming, the most frequent one is *naming*[29]. One reason as to why that is might be that naming (as language) is an inherently social activity[30] and therefore a name is an utterance which only makes sense when done in the expectation of someone else's comprehension of that name. This is supported by the fact that the process of creating a variable or function name on one's own is often more time-

[29] (attributed to Phil Karlton), link

[30] Volosinov, V. N., Marxism and Philosophy of Language, Harvard University Press, 1986.

consuming when done alone[31], as reported by developers. Naming, furthermore, aims not just at describing, but at capturing the essence of an object, or of a concept. This is a process that is already familiar in literary studies, particularly in the role of poetry in naming the elusive. For instance, Vilém Flusser sees poetry as the briging-forth that which is conceivable but not yet speakable through its essence in order to make it speakable through prose[32], using the process of naming through poetry in order to allow for its use and function in prose. In this light, good, efficient and beautiful names in code are those who can communicate the essence of the concept that is being communicated (or parts thereof).

On a purely sensory level (visual and auditory), surface-level aesthetic criteria related to naming are that of **character length** and **pronounceability**. Visually, character length can indicate the relative importance of a named concept within the greater structure of the code-text. Variables with longer names are variables that are more important, demand more cognitive attention, offer greater intelligibility in comparison with shorter variable names, which only need to be "stored in memory" of the reader for a smaller amount of time. These visual cues, again, alleviate cognitive pain when trying to understand code, and therefore hold greater aesthetic value when respected. Pronounceability, meanwhile, take into account the basic human action of "speaking into one's head" and therefore participates in the requirement for communicability of source code amongst human readers. Similar to proper indentation and typographic consistency (see above), this particular criterion exists in the category of aesthetic criteria which are required, but not sufficient, for beautiful code.

Equally visual, but aesthetically pleasing for typographical reasons, is the **casing** of names. Dealing with the constraint that variable names cannot

[31] Particularly, it is interesting to note that functions are easier to name than variables.

[32] Flusser, Vilém, On Doubt, Univocal Publishing, 2015.

have whitespace characters as part of them, casing has resulted into the establishment of conventions which pre-exists the precise understanding of what a word denotes, by first bringing that word into a category (all-caps denotes a constant, camelCasing denotes a multi-word variable and first-capitalized words indicate classes or interfaces. By using multiple cues (here, typographical, then semantical, as explicited below), casing again helps with understandability. Furthermore, casing, by its existence as a convention, implies that it exists within a social community of writers and readers, and acknowledges the mutual belonging of both writer and reader to such a community, and turns the code-text from a *readerly* text further into a *writerly* one[33].

Following these visual, auditory and typographical criteria, an aesthetically-pleasing vocabulary is a vocabulary which strictly names **functions as verbs and variables as nouns**. In the vein of making a correspondance between machine language and human language, there is here a clear mapping between the two: functions *do* things and variables *are* things. If it’s the other way around, while respecting the criteria for consistency, functions as nouns and variables as verbs hints at what it is not, are counter-intuitive and ultimately confusing—confusion which brings ugliness. The noun given to a variable should be a hint towards the concept addressed, and ideally address *what* it is, *how* it is used, and *why* it is present. Each of these three aims aren’t necessarily easily achieved at the same time, but finding one word which, through multiple means, points to the same end, is an aesthetic goal of source code writers. Particularly, limiting the naming to be the answer to only one of those questions (only *what*, only *what*, or only *why*) could potentially confuse the reader more than it would enlighten her. A beautiful name is a name which differentiates between *value* (obvious, decontextualized, and therefore unhelpful, as seen by the general frowning-upon of using *magic numbers*) and *intention*,

[33] Barthes, Roland, Le Plaisir Du Texte, Seuil, 1973.

informing the reader not just about the current use, but also about future possible use, in code that is written or yet to be written. We see here a paradox between direct conceptual relationship between a name and what it denotes, and the multiple meanings that it embodies (its description, its desired immediate behaviour, and its purpose). Such a paradox is however overcome in the community of code poets.

While, in the community of software developers, variable names should then have a 1:1 mapping with the object or concept they denote, this isn't the case in other communities, whether those that rely on obfuscation, in which confusion becomes beautiful, or in poetic code, in which **double-meaning** brings an additional, different understanding which ultimately enriches the complexity of the reading[34]. Indeed, it is the easiest way for writers to offer metaphors, and provides an entry point in to the possibility that all source code is itself a *practical metaphor* for the task—and therefore the problem—at hand. This aesthetic criteria of double-meaning comes from poetry in human languages, in which layered meanings are aesthetically pleasing, because they point to the un-utterable, and as such, perhaps, the sublime. The way that this community (code poets and artists) address the aesthetic problem of naming and, more generally, how source code and literature make use of metaphors, is part of the next steps that will be taken in this research.

A final aesthetic criterion for vocabularies is the **limitation of function arguments**, according to which arguments given to a function should be either few, or grouped under another name. Going back to the structural criterion above of limiting input/output and keeping groups of statements conceptually independent, function arguments solves this requirement at the level of vocabulary, demonstrating in passing the relative porosity of those categories. Indeed, the naming of variables also reveals the pick of

[34] Knuth, Donald, Literate Programming (Lecture Notes), Center for the Study of Language and Inf, 1992.

adequate data-structures, echoing those who claim that the data on which the code operates can never be ignored, and that beautiful code is code which takes into account that data and communicates it, and its mutations, in the clearest, most intelligible, possible way.

- Comments

Comments in code do not seem to fall clearly in any of the three categories above. By definition ignored by the compiler/interpreter, comments can be erroneous statements which will persist in an otherwise functional codebase, and are therefore not trusted by experienced, professional software practicioners. In this configuration, comments seem to exist as a compensation for a lack of functional aesthetic exchange. By functional aesthetic exchange I mean an exchange in which a skilled writer is able to be understood by a skilled reader with regards to *what* is being done and *how*. If any of these conditions fail (the writer isn't skilled enough and relies on comments to explain what is going on and how it is happening, or the reader isn't skilled enough to understand it without comments), then comments are here to remedy to that failure, and therefore are an indicator (but, again, not a proof) of non-beautiful code. For instance, referencing a variable name in a comment is a sure indicator of a message which refers to the what/how of a group of statements and is on the verge of stating the obvious (if not already stating the obivous).

The situation in which comments seem to be tolerated is when they provide contextual information, therefore (re-)anchoring the code in a broader world. For instance, this is achieved by offering an indication as to *why* such an action is being taken at a particular moment of the code, called *contractual comments*, pointing at the social existence of source code. This particular use of comments seems to bypass the aesthetic criteria of code being self-explanatory, but nonetheless integrates the criteria of code being writable, a piece of code which, by its appearance, invites the reader to contribute to it, to modify it. As such, in an educational setting (from a classroom to an open-source project), comments are welcome, but rarely quoted as criteria for beautiful code, which seems to indicate that the appreciation of beautiful code does require a certain level of skill.

Aesthetics as a purposeful, functioning device

This set of criteria is only the first of multiple (including those of artists, hackers and academics), and is intended to be limited to the community of practicioners it stems from, as well as inform a consolidated set of principles which could possibly apply to any piece of source code. While the content of the framework seems to apply in a broad to any commentary on source code found in the gathered corpus of software developers, its organization in structure/syntax/vocabulary mirror a parallel structure in the aesthetic *experience* of the reader of a code-text. Such an aesthetic experience could be organized in terms of cognitive depths: reading (e.g. code is properly formatted, can be read), understanding (e.g. code communicates what it does), enlightening (code communicates more than what it does). This parallel structure also reflects the important fact that reading code is a different process than writing code. Indeed, while writing code can have similarities with writing prose or poetry, reading code, on the other hand, is more akin to investigative work and dissection than to leisurely skimming over a novel. Further research on this is needed, particularly along the axes of linear/non-linear reading, the requirement (or not) of paratexts, as well as reader positions, which then ties back into the existence of social contexts of aesthetics.

At this point, this set of criteria for software developers confirms a close relationship between beauty and understanding (i.e. *is beautiful that which I can easily understand and work with*). Preliminary examination of the other groups of software practicioners also point to the presence of understanding as a writer and a reader engage with a code-text. For instance, for hackers, *is beautiful that which challenges understanding, that which is not understandable*, either by removing agency, or by purposefully not assuming agency. For academics, *is beautiful that which provides an understanding beyond what is immediately there*, and opens up deeper insights into theoretical realms of computation, algorithms or language design. For artists, *is beautiful that which offers a different, subjective understanding through poetic interpretation*. Each of these approaches will be the subject of further research this semester.

Following this, arises the question of whether a similar set of aesthetic criteria can be used to elicit multiple kinds of understanding, or if separate kinds of understandings require separate kinds of aesthetic criteria? If providing the understanding of something is akin to the act of *making clear* of that thing, then one could see *the art of programming* as the art of the obvious, of the transparent. This requirement to make intelligible would go against certain definitions of art as a self-sustaining, self-referential practice, and would locate code between an art and a craft, perhaps even blurring the boundary between them. A piece of code which has to involve some concept to be understood implies that writing code is what I would call a *functional aesthetic practice*, an aesthetic practice which needs to *do* something in order to be appreciated as such. It could then help to reconsider the separation between arts and craft, and perhaps seeing art as something which necessarily deals with the addition and/or modification of understanding? A similar problem about this dichotomy between art and craft can be seen in architecture is in a similar situation, further supported by the similarities in physical architecture and software architecture. Between craft and practice, architecture is first and foremost meant to be used; and architecture can also elicit similar (multiple) understandings.

A further implication would touch upon artistic practices in general. Do all aesthetic objects communicate some knowledge that is to be understood by the audience? And therefore, can it be said that a good artwork is an artwork which reaches beyond itself, and always refers to the “problem domain” (a metaphor for the “real world” in programming)? In order to approach this question, we must first inspect the relationship between code and literature.

Code and Literature

The relationship between writing and reading code and reading and writing literature, while much[35] written[36] about, doesn’t however seem immediately obvious, beyond the fact that both use words as their raw material to communicate concepts. Indeed, most literary theorists and humanities scholars focus on the literary implications of *executed code*, while often overlooking the potential aesthetic renewals which could stem from a closer examination of source code. On the opposite, computer science practicioners have indeed claimed similarities between code and literature, often as statements within elaborate justification. Why exactly computer scientists do so could stem from reasons such as: legitimizing their field through connections with additional fields, pointing out the similar activities of writing and reading, highlighting the fact that their activity is not just a scientific endeavour. However, while no consistent theory of source code as a *subset* of literature exists today, I would rather look at it as two separate endeavours, which nonetheless:

- use similar materials (languages, or coherent systems of syntactic tokens)
- involve similar processes and approaches (blank pages, drafts, attention to details, recourse to imagination)

[35] Matsumoto, Yukihiro, Treating Code As An Essay, in Beautiful Code, op. cit.

[36] Knuth, Donald, Literate Programming (Lecture Notes), Center for the Study of Language and Inf, 1992.

- overcome similar problems (need to clarify complex cognitive structures, lack of clear communication between two subjective minds)
- and desire to achieve similar ends (share an understanding of a broader concept).

These two endeavours can be placed along a gradient between personal interpretation and objective efficiency. Where different manifestations of writing code or writing literature stand depends on what they attempt at representing, at making understandable. For instance, if the aim is to make understandable the operations of a hardware timer, then the interpretation must be of the strictest nature[37]. The writing must also be subject to that requirement, in order to prevent any confusion in the reader and, ultimately, faulty usage with practical consequences. Further along that gradient, if the aim is to make understandable the implementation of a high-level algorithm, or pattern (say, a regular expression search[38]), there is a little more room for an interpretative *approach* towards an understanding of the idea of a regular expression, providing some agency to the reader to form a *comprehensive* mental model of a regular expression, rather than reading and assimilating an *exhaustive* one.

On the other side of this gradient, closer to the interpretation, that which is to be communicated is part of human endeavours (e.g. employee management systems, user-facing software, and more generally the kind of software which has hugely benefited from the OOP paradigm). Since part of the complexity of software resides in the necessity to encode discreetely that which is continuous, code gets closer to literature, drawing further from metaphors and narration in order to facilitate understanding. It does so as it inherently leaves room for uncertainty, for interpretation and for imagination with regards to what a user could do or would do, hinting at

[37] Neville-Neil, George V., Beautiful Code Exists, If You Know Where To Look, ACM Queue, Vol. 6, Issue 5, 2008.

[38] Beautiful Code, op. cit.

broad themes and possibilities of action, on top of the strict operation of its own code. Beautiful code is code which allows this personal interpretation while leaving no doubt as to what it is currently doing.

The distinction that some of the more technically-savy practicioners operate between code and comments illustrates this relationship quite well with the statement that *code never lies, comments do*. Comments, by virtue of being completely ignored by the compiler or interpreter, are the component of source code that is closest to prose. To what extent are comments helpful because they lie, or because of their unreliable nature? Comments are only necessary when one wants to explicit *why* a section of code does something. The question *why* hints at broader, more metaphysical concerns, concerns that are harder to communicate through code, and easier to communicate through literature. Another example is that of Donald Knuth's work *literate programming*[39], which is a set of languages, tools and practices which attempts at establishing comments as the canonical source of any compilable or interpretable source code. The aim is to allow the writer to write in "plain English", and then generate source code from this description. Such an approach both claims that there are ties between writing source code and writing texts, yet crystallizes the difference between both. Effectively, a `.web` file (to take the example of the first literate programming processor) is but a markup language which weaves in both natural and machine languages but does only so superficially, ultimately splitting each of those languages in separate files (`.tex` and `.pas`, respectively). I would therefore argue that, given fact that literate programming can be seen as *inserting code within literature*, code and literature are distinct practices which nonetheless have the potential to mutually inform each other, as word-based crafts.

If there are indeed parallels that can be made between code and literature, without claiming that code *is* a new form of literature, to what extent does

[39] Knuth, Donald, Literate Programming, op. cit.

code and literature use the same techniques (of conceptual distancing, naming, etc.), and to what end? One approach would be to compare both practices in terms of **metaphors**: how they are used (to clarify or to mystify), and to what purpose (i.e. for which kind of understanding). Furthermore, it seems that, just as there isn't a single kind of code, there also isn't a single genre of literature. While examining parallels, it will be useful to acknowledge that code can draw from styles such as minimalism, absurdist fiction or technical/scientific writing. Without a single monolithic vector against which the beauty of a code-text is evaluated, this would assess such aesthetics along different axes, between craft, literature and engineering.

next steps

The work so far has established main aesthetic criteria for a particular kind of software practicioners: software developers constituting the vast majority of people writing and reading source code. Through this endeavour, I have highlighted the role that these criteria have with regards to understanding source code, in terms of reducing cognitive friction and facilitating more or less strict interpretations of the concepts and ideas that are being constructed and communicated by the writer. These preliminary findings support a *functional approach to aesthetics*. This implies that aesthetics can (and, in this specific context, should) have a transitive role: if the concept that they represent isn't effective, or interesting or novel to the reader, then the code will not be considered "beautiful". The goal here is that code becomes "self-documenting", "transparent", in that it doesn't bring too much attention to itself, but rather points to the problem that it is attempting to solve by showing what it does, how it does it and why. This corresponds to Nelson Goodman's theory on the symbolic meaning of art[40], as he argues that works of art, as an aesthetic manifestation, always communicates concepts to the viewer/reader, facilitated through formal

[40] Goodman, Nelson, op. cit.

cues through the use of denotation, connotation and representation. The practice of writing source code could therefore support a conception of works of art as useful and functional communication tools.

These cues, organized under the distinction of structure/syntax/vocabulary, can provide different depths of understanding, but it isn’t yet determined as to whether it can provide different *kinds* of understandings, and whether the same set of critieria can help understand different concepts (between say, hardware architecture, algorithm implementation, real-world situation or poetic imagination). The immediate next aspect that I will be focusing on is therefore to highlight the aesthetic criteria which support two other communities of practice: hackers and artists. This will help identify to what extent aesthetics can operate across communities of practice, or to what extent these communities of practice operate under independent aesthetic standards. Particular attention will be paid to how each of these communities involve metaphors in their work (under what form and for which purpose). This will involve close readings of selected source code poems one one side and analysis of accounts and descriptions of “hacks” on the other side. The research questions will remain the same: what are practicioners trying to make understandable? and what aesthetic mechanisms are they using to do so?

Another direction in which to answer the potential cross-domain nature of aesthetic critieria is that of programming languages. I intend to further analyze the perceptions around beauty and languages. So far, it seems that there are some languages that are considered more beautiful than others (e.g. Ruby, Lisp vs. Java, JavaScript), and the reasons why that is could give further insight into the nature of aesthetic criteria in source code, particularly in further defining *“elegance”* and *“clarity”*, broad terms often employed by practicioners to describe beautiful code. The hypothesis here is that some languages are more prone to support aesthetic criteria for beautiful code, both intrinsically (through idiomatic features) and extrinsically (due to external discourses and paratexts confirming this “beauty”).

Finally, there needs to be additional research undertaken regarding metaphors, and how they relate to mental models and knowledge representation. While a broad field, I intend to approach it through literary theory, as well as through the small field of programming psychology. As explicited above, it seems that code and literature are only loosely related, more through the process of writing than the process of reading. The abundant use of metaphors in literature, and the essential quality of code to execute commands which stand for something else (e.g. keywords such as `open`, `close`, `listen`, `break`, but also vocabularies created by the writers while naming variables, functions, classes) would be a fruitful terrain for a comparative examination. Such a comparison would help in defining exactly what are the roles of metaphors in writing and reading code, how they are manifested through aesthetics and how they can illustrate the similarities and differences between code and literature.

final questions

- how should i approach metaphors? from a philosophical stand, a psychological one or a literary one? are there any references that I should be reading or looking at?
- should i also look into what metaphors are used to make sense of code? that is, metaphors that describe the act of writing/reading code itself, on top of metaphors which are used *in* the code