Critical Hermeneutics, 4(1), 2020
Biannual International Journal of Philosophy
http://ojs.unica.it/index.php/ecch/index
ISSN 2533-1825 (on line); DOI 10.13125/CH/4317

Received: 8/3/2020
Accepted: 25/8/2020
Published: 9/10/2020

Digital Reflective Judgement: A Kantian Perspective on Software

Luca M. Possati

Abstract

In this paper, I formulate an analysis of software from a Kantian perspective. The central thesis is that software is a form of reflective judgment, namely, "digital reflective judgement". This transcendental approach allows us to overcome the limitations of an overly dualistic and over-intellectualized conception of software. The paper is structured as follows. In section 2, I develop a series of criticisms of Turner's (2018) approach. Turner defines software as a computational artifact and distinguishes two series of its properties: functional and structural. I argue that this distinction cannot be applied to software and that Turner's approach cannot explain the essence of software, namely, its twofold nature—abstract and concrete—at the same time. Moreover, Turner's perspective is characterized by some philosophical limitations. In sections 3 and 4, I present a proposed definition of software from a transcendental Kantian perspective, that is, by using the concept of reflective judgment. In section 5, I explain why and how we can consider software as a new form of reflective judgment. This judgement is based on a specific type of imaginative act that mediates between physical implementations and mathematical structures. In section 6, through a parallelism between software and the Kantian judgment of taste, I hold that the condition of possibility of software is the principle of finality, which is shown in the design. Software is, above *all, a design act. In the conclusion, I show why this approach overcomes Turner’s limitations and is much closer to how programmers conceive their work.*

Keywords: Software, Reflective Judgement, Kant

1. Introduction

The central issue of this paper can be described in the following way:

> A central topic in the philosophy of computer science concerns the ontological status of programs. While algorithms are generally taken to be mathematical objects, the nature of programs is less clear, although semantic concerns are central: their ontological status is closely allied to their semantic status. In particular, a semantic account of programming languages is taken to involve a machine of some kind. But what kind of machine? Is it abstract or concrete? If a physical machine is taken to fix the meaning, then semantically and ontologically, programs are primarily physical devices. Conversely, if an abstract machine is employed, then programs are abstract in nature. However, the nature of programs is not so easily and neatly settled: both abstract and physical devices seem to be involved (Turner 2018: 15).

The central thesis of this paper is that the ontological issues related to software have to be re-formulated from a Kantian perspective. I propose a thinking of software as a form of reflective judgment. I claim that this transcendental approach will allow us to overcome the limitations of a dualistic and over-intellectualized conception of software as well as understand other dimensions of software that a merely technical explanation invariably misses. My goal is not to give a comprehensive definition of software. I think that software is an overly elusive and complex phenomenon, and a univocal definition inevitably makes us lose its multistability (see Ihde 1990). Instead, my goal is to define the conditions of a philosophical analysis of software as a way of understanding this multistability.

The paper is structured as follows. In section 2, I develop a series of criticisms of Turner's (2018) approach. Turner defines software as a computational artifact and distinguishes two series of its properties: functional and structural. I argue that this distinction cannot be applied to software and that Turner's approach cannot explain the essence of software, namely, its twofold nature—abstract and concrete—at the same time. Moreover, Turner's perspective is characterized by some philosophical limitations.

Consequently, in section 3 and 4, I present a proposed definition of software from a transcendental Kantian perspective, i.e., by using the concept of reflective judgment. Turner fails to solve the question of the ontological status of software because of the overly strict manner in which he connects ontology and semantics. Reformulating the ontological question in transcendental and aesthetic terms gives us the possibility of escaping Turner's limitations and explaining the plastic and dynamic nature of software.

In section 5, I explain why and how we can consider software as a new form of reflective judgment. My claim is that this judgement is based on a particular type of imaginative act that mediates between physical implementations and mathematical structures. This imaginative act is based on some synthetical structures, i.e., specific forms of writing. Software programming is essentially an art of writing.

In section 6, through a parallelism between software and the Kantian judgment of taste, I hold that the condition of possibility of software is the principle of finality, which is shown in the design. Software is, above all, a design act. In the conclusion, I show why this approach overcomes Turner's limitations and is much closer to how programmers conceive their work.

2. The Problem of the Synthesis

According to Turner (2018),

> Languages and machines represent the two ends of the computational spectrum: the abstract and the physical. They come together at the digital interface, the very lowest level in the computational realm. Digital circuits are employed to store, communicate, and manipulate data. Flip-flops, counters, converters, and memory circuits are common examples. Their building blocks are called gates, the most central of which correspond to arithmetic and Boolean operations. These are simple logic machines, so named because they are intended to represent some form of numerical or Boolean operation. More complex machines are built from them by connecting and composing them in various ways, where the most general-purpose register-transfer logic machine is a computer.

Even though Turner's book remains an important reference point in computer science philosophy, this view appears to be oversimplified. I contend that Turner does not grasp the complexity of the philosophical problem underlying his premise. Let us try to correctly formulate this problem: how is the a priori synthesis of Boolean operations and machines possible? This synthesis is a priori because it is the condition of possibility of all our digital experiences and is independent of these experiences (the computer also acts in my absence and performs these operations). Moreover, what does the synthesis between the machine, Boolean operations, and user experience guarantee? How can we know that this correspondence takes place?

Boolean logic truth tables give us a functional description of a digital circuit. This description is formulated in a formal language and says how the digital circuit should work. However, the digital circuit is also a physical object, i.e., a set of material and electrical pulses. As a physical object, the electrical circuit has a series of structural properties that tell us what it is and what it does (Turner 2018, 33), but the functional and physical properties are incompatible.[1] In fact, the functional properties "provide no account of how the actual electronic devices are to be built; they do not describe how the computations are to be carried out. They are functional specifications, not [physical] ones, and they cannot be easily turned into the latter. They tell us what the actual physical device should do: the what not the how" (Turner 2018: 33). Functional and physical properties are also incompatible with the characteristics of user experience, which incidentally may have knowledge of neither of them.[2]

In chapter five of his book, Turner talks about software ontology. He distinguishes between functional description (specifications), structural description (HL languages), and physical description (implementation). According to him, "in the case of programs,

[1] "Technical artifacts are, at least *prima facie*, always physical objects, but they are also objects that have a certain function. Looked upon merely as physical objects they fit into the physical or material conception of the world. Looked upon as functional objects, however, they do not. The concept of function does not appear in physical description of the world; it rather belongs to the intentional conceptualization. Technical artifacts thus have a dual nature: they cannot exhaustively be described within the physical conceptualization, since this has no place for their functional features, nor can they be described exhaustively within the intentional conceptualization, since their functionality must be realized in an adequate physical structure" (Kroes and Meijers 2002).

[2] Turner talk also about "structural properties" defined by a designer using another specific formal language.

implementation is a mechanism that, given a symbolic program as input, returns a physical process" (49). As computational artifacts, programs mediate between functional and physical properties, yet Turner fails to explain how.

He does not see a problem in this transition from the symbolic to the physical. Software is a complex abstract structure, made up of many levels and languages and levels of abstractions, which is capable of producing a physical effect. It is a language that does what it says. It has a performativity that is independent of any human intervention. How, therefore, is it possible for a symbolic apparatus to produce a physical effect? How can an abstract mathematical structure (see Indurkhya 2017) implement physical operations? Turner limits himself to quoting Colburn's (1999) well-known thesis about "harmony", according to which there is a "fundamental harmony" between the physical and the symbolic in the program. However, this is not an explanation at all. In Colburn's vision, the programmer appears like the *deus ex machina* that harmonizes the symbolic and the physical, allowing the machine to function. Colburn (1999: 17) speaks of "pre-established harmony", namely, a parallelism between the code and machine, established not by God but by the programmer. "Programmers today can live almost exclusively in the abstract realm of their software descriptions, but since their creations have parallel lives as bouncing electrons, theirs is a world of concrete abstractions" (18). Turner seems to actualize Colburn's solution by saying that, "presumably, it is via the semantics that the programmer is able to design the program from the specification, and it is via the semantics that the programmer is able to explain why and justify the claim that the program meets the specification" and, so, the implementations (Turner 2018: 51). He distinguishes three levels—syntax, semantics, and implementation—and thinks that the passage from the first to the third is due to the second.

Let us consider Turner's (99–100) example. We have the following formula:

A: = 13 + 74

This string of code can receive a "physical interpretation" that will have the following form:

physical memory location A receives the value of physically computing 13 plus 74.

However, what does "physical interpretation" mean? In reality, this interpretation is just another linguistic formulation of that string of code, i.e., its translation into another language. That string of code corresponds to a certain state of the CPU and, therefore, to a series of operations, i.e., physical states—what the machine actually does. The correspondence is ensured by the programmer's semantic choices. Nevertheless, as Turner claims, a semantic explanation concerns only the functional level; it cannot tell us anything about the physical operations of the machine. A physical explanation concerns only the structural and physical levels, i.e., how the machine effectively acts, and it cannot tell us anything about how the machine should act. Neither the functional nor the structural levels have the resources to explain their connection. In other words, the connection sought can be neither functional nor structural nor physical. Turner (101) limits himself to saying that "the physical implementation is subject to the abstract interpretation, and the meaning of the construct is given by the abstract account alone". He still does not see the complexity of the problem of the connection of levels or of the relationship between the different levels and user experience.

I make four fundamental criticisms of Turner's approach:

- He does not consider the problem of the interaction, i.e., the relation with user experience;

- The basis of his theory is an ontology of the thing and not of the process—for this reason, he defines software as an object, an artifact—but I do not think that this is appropriate to explain software;
- His is an excessive intellectualization of software.

Let us try to clarify the problem. We have three distinct levels:

The functional level → formal syntax and semantics

The physical level/1 → implementation/problem-solving

The physical level/2 → user experience.

Neither a monistic solution (all levels can be reduced to one; they form a unitary whole) nor a dualistic solution (the physical and abstract are two distinct and non-communicating levels) appear satisfactory. If we choose the first solution, we have to explain the differences between these levels. If we choose the second way, we must explain the unity of these levels. Turner chooses the second way, which is the reason he is forced to invoke an inexplicable "harmony". He also refers to Landin's correspondence principle (170–171), but this changes nothing. Separating the functional level from the physical level is only an intellectual abstraction. All the levels are connected in the digital experience: user experience is a continuum. When I use my laptop, I do not have many different experiences—one of the functional level, another of the structural level, another of the physical machine, etc. My digital experience is uniform. Thus, my question is: how is this continuum possible? My belief is that we need to re-formulate the ontological question of software in a transcendental way. In the next sections, I will try to develop this approach by following the analogy between software and the Kantian reflective judgment.

3. The Kantian Reflective Judgment

According to Kant, a "judgment" (*Urteil*) is a specific kind of "cognition" (*Erkenntnis*), i.e., a conscious mental representation of an object (*Critique of the Pure Reason*, A320/B376).[3] This representation has a synthetic form: it unifies and organizes raw, unstructured sensory data according to universal concepts, rules, or principles. Judgement is essentially the faculty of thinking of the "particular" (the representation of a singular thing) as being contained under the "universal" (the general representation). This synthesis is the characteristic output of the "power of judgment" (*Urteilskraft*). The power of judgment is a cognitive "capacity" (*Fähigkeit*), more specifically, a *spontaneous* and *innate* cognitive capacity. By virtue of this, it is the "faculty of judging" (*Vermögen zu urteilen*) (A69/B94), which is also the same as the "faculty of thinking" (*Vermögen zu denken*) (A81/B106). It is a controverted question whether, according to Kant, there is only one kind of synthesis or many different kinds. Moreover, terms such as "spontaneity" or "concept" can have different meanings in Kant's works. These issues are closely linked to the recent debate about Kant's conceptualism vs. Kant's non-conceptualism in relation to his theory of judgment and the ensuing implications for interpreting and critically evaluating his transcendental idealism and the "Transcendental Deduction of the Pure Concepts of the Understanding" (see Hanna 2001, 2006, 2005, 2017; Land 2011, 2015, 2016; Ginsborg 2006). However, I do not want to tackle these issues here.

I want to stress three points. First, for Kant, judgments are essentially propositional cognitions—from which it immediately follows that rational humans are, more precisely, propositional

[3] I quote using the relevant volume and page number from the standard "Akademie" edition of Kant's works: *Kants Gesammelte Schriften*, edited by the Königlich Preussischen (now Deutschen) Akademie der Wissenschaften (Berlin: G. Reimer [now de Gruyter], 1902–).

animals. The connection between judgment and language is, therefore, essential.

Second, Kant distinguishes the logical form and the propositional content of a judgment. The logical forms are summarized in the "table of judgements" (*Critique of the Pure Reason*, A52–55/B76–79). The propositional contents, which are more fundamental than the logical forms, are classified according to two conceptual couples: a priori/a posteriori, analytical/synthetical.[4] Briefly, the propositional content of a judgment can vary along at least three dimensions: (1) its relation to sensory content, (2) its relation to the truth-conditions of propositions, and (3) its relation to the conditions for objective validity.[5]

Third, Kant distinguishes between propositional contents and *the use of* propositional contents. It is possible for a rational subject to use the same propositional content in different ways. What does this mean? The fundamental difference is that between (a) theoretical use and (b) non-theoretical use. The first use aims to formulate true propositions about the world in order to obtain some knowledge, i.e., science. The second use does not aim to formulate true propositions about the world; thus, its aim is pragmatic, moral, aesthetic, or teleological.

To specify the distinction between theoretical and non-theoretical

[4] For the meaning of these expressions in Kant, see Eisler (1994: 48–54). For Kant, there are three types of judgment: analytical a priori, synthetic a posteriori, and synthetic a priori (see Eisler 1994, 585ss). The supreme principle of all synthetic judgments is that "every object is subject to the necessary conditions of the synthetic unity of the different intuitions in a possible experience. […] The conditions of the possibility of experience in general are at the same time conditions of the possibility of the objects of experience, and for this they have an objective validity in a synthetic judgment a priori" (AK III 39-40); "All analytical judgments rest entirely on the principle of contradiction and are by nature a priori knowledge […]" (AK IV, 266-267).

[5] I am aware that this description is schematic and does not show the real complexity of Kant's thought on this topic. However, what interests me in this section is to highlight only the fundamental points of Kant's theory of judgment and then focus on the distinction between determinant and reflective judgment.

uses of propositional contents, Kant introduces the distinction between "determining" judgment and "reflective" judgment. In the first "Introduction" to *The Critique of Judgment* (1790), he writes:

> Judgement in general is the faculty of thinking the particular as contained under the universal. If the universal (the rule, principle, or law) is given, then the judgement which subsumes the particular under it is determining. This is so even where such a judgement is transcendental and, as such, provides the conditions a priori in conformity with which alone the subsumption under that universal can be effected. If, however, only the particular is given and the universal has to be found for it, then the judgement is simply reflective (Kant 2016: 53).

Here, Kant develops some remarks about the regulative use of the ideas of reason, which appeared in the first *Critique*'s *Appendix* to the "Transcendental Dialectic", in particular, the distinction between "apodictic" and "hypothetical" uses of judgment (A647/B675). The difference between the determining and reflective uses of a judgment has to do with the way in which the synthetical structure of the judgment is interpreted. The determining use presupposes a high-order representation under which to subsume the particular. It determines an individual or narrower concept by using a given general "determinable" concept or principle. The reflective use follows the opposite way. It presupposes a particular individual or narrower concept and advances from it toward a universal or more general concept. Thus, the reflective judgment directly invokes the cognitive subject's ability to form higher-order representations through the act of reflection (*Überlegung)* and, consequently, to be rationally self-conscious or apperceptive. The aesthetic judgment (the judgement of taste) and the teleological judgment are expressions of the reflective use of propositional contents. It is worth reading the entire passage from the first "Introduction" to *The Critique of Judgment*:

> The determining judgement determines under universal transcendental laws furnished by understanding and is subsumptive only; the law is marked out for it a priori, and it has no need to devise a law for its own guidance to enable it to subordinate the particular in nature to the universal. But there are such manifold forms of nature, so many modifications, as it were, of the universal transcendental concepts of nature, left undetermined by the laws furnished by pure understanding a priori as above mentioned, and for the reason that these laws only touch the general possibility of a nature (as an object of sense), that there must needs also be laws in this behalf. These laws, being empirical, may be contingent as far as the light of our understanding goes, but still, if they are to be called laws (as the concept of nature requires), they must be regarded as necessary on a principle, unknown though it be to us, of the unity of the manifold. The reflective judgement which is compelled to ascend from the particular in nature to the universal stands, therefore, in need of a principle. This principle it cannot borrow from experience, because what it has to do is to establish just the unity of all empirical principles under higher, though likewise empirical, principles, and thence the possibility of the systematic subordination of higher and lower. Such a transcendental principle, therefore, the reflective judgement can only give as a law from and to itself. It cannot derive it from any other quarter (as it would then be a determining judgement). Nor can it prescribe it to

> nature, for reflection on the laws of nature adjusts itself to nature, and not nature to the conditions according to which we strive to obtain a concept of it – a concept that is quite contingent in respect of these conditions (Kant 2016: 23).

Aesthetic and teleological judgments can only be reflective: they do not produce knowledge.[6] While determining judgment is based on several a priori principles, i.e., the principles of pure reason (transcendental logic), which are the basis of objective knowledge, reflective judgment has only one a priori transcendental principle—finality—which is universal and subjective at the same time. Therefore, reflective judgment interprets the particular case according to finality. It can have two forms: 1) in the first case, the principle of finality is applied to the relationship between the subject and the representation of the particular case and, therefore, to the spontaneous agreement between imagination and understanding, which produces a delight—interpreted as a sign of finality; 2) in the second case, finality is applied to the organization of nature through understanding and reason. In this case, it can only be subjective because it cannot be the object of a possible experience, i.e., a phenomenon. Through the faculty of judgement, nature is represented as if a supreme intelligence had arranged the unity of all empirical laws.

However, the principle of finality is not to be confused with practical finality, as we can read in the first "Introduction" to *The Critique of Judgment*:

[6] One must not make the mistake of thinking that the *Critique of Pure Reason* deals only with determining judgment and, instead, that the *Critique of the Judgment* deals only with reflective judgment as if, in Kant, there was a clear distinction between these two uses of propositional contents. It must be emphasized that Kant, in the third *Critique*, defines aesthetic and teleological judgments as *only* reflexive in the sense that these judgments are *entirely* reflective. Many other judgments are determining and reflective at the same time (Longuenesse 1993: 208–215).

> […] this transcendental concept of a finality of nature is neither a concept of nature nor of freedom, since it attributes nothing at all to the object, i.e., to nature, *but only represents the unique mode in which we must proceed in our reflection upon the objects of nature with a view to getting a thoroughly interconnected whole of experience, and so is a subjective principle, i.e., maxim, of judgement*. For this reason, too, just as if it were a lucky chance that favoured us, we are rejoiced (properly speaking, relieved of a want) where we meet with such systematic unity under merely empirical laws: although we must necessarily assume the presence of such a unity, apart from any ability on our part to apprehend or prove its existence (emphasis added) (Kant 2016: 46).

The principle of finality is the essence of the faculty of judgment as an autonomous faculty with respect to understanding and practical reason. In the principle of finality, the faculty of judgment gives itself a law in order to think about the unity of nature. The faculty of judgment presupposes this law in order to obtain an overall view of nature that is acceptable to us. Therefore, finality has a hypothetical nature. In *Logik* (§81), Kant describes finality as the "analogon" of the logical universality. Thus, the faculty of judgment produces inductive and analogical reasoning.[7]

Even if it does not have a cognitive function, reflective judgment plays a crucial role in science from a heuristic and methodological point of view. In the third *Critique*, Kant emphasizes the need for

[7] The problem of analogy in Kant is very complex. I do not want to tackle this issue here. In Kant, there are several ways in which the term "analogy" is used. I would say that we can distinguish three main meanings: theological, cognitive (the analogies of experience), and mathematical. See Callanan (2008).

teleological judgment for the study of biology. If the understanding explains a coherent physical science based on universal laws, it is not sufficient to explain the smallest and simplest living organisms. The life of a worm or the growth of a blade of grass can never be understood starting from a determining judgment; it can only be understood through reflective judgment. Notions such as "gender" or "species" have a heuristic and methodological value to the extent that they are used in connection with determining judgment. They cannot be the basis of synthetic a priori judgments, but they can help in explaining what cannot be stated or formulated in synthetic a priori judgments.

4. The Digital Reflective Judgement

I propose that software be conceived from a Kantian perspective, i.e., as a kind of reflective judgment. I call this new form of reflective judgment "digital reflective judgment" (henceforth DRJ). Why is it that software cannot be compared to determining judgement? It cannot be so compared because its function is not limited to subsuming a concrete problem to an abstract computational structure, namely the Turing machine. Software is not a mechanical process.

Let us look closely at *what the programmer does*. Like reflective judgment, software starts from an individual case, i.e., the problem to be solved. Software is essentially problem-solving. In order to solve a particular problem, the programmer creates a solution, which should be universal, i.e., applicable to all such problems. The programmer has not yet made categories that can be used to subsume the particular case/problem. She/he has to invent and create these categories. This universal is the set of syntactic and semantic categories (the high-level language) that the programmer uses to define and solve a particular problem. Therefore, the first work of the programmer is to "translate" the particular problem (drawing, writing, printing, etc.) into an abstract structure. This is not automated work. The programmer must choose the language that best works in relation to the problem she/he faces—for instance, building a website, a smartphone app, or a calculation program or creating a data visualization, etc.—as well as the libraries and best data architecture. This is creative work.

Now, in Kantian terms, the process that leads from the individual problem to its re-interpretation in a formal system is completely non-theoretical. In fact, this process adds nothing to the "heart" of the computer, namely, the Turing machine—the computability. Using a certain programming language or a certain algorithmic style does nothing to influence the behavior of the Turing machine. The Turing machine does not understand the high-level language, the particular problem, or human reality. The programmer reflects on the problem and proposes an algorithmic solution expressed in the high-level language. She/he provides an interpretation of the problem and its solution (the higher-order representation under which to subsume the particular). This solution has to be translated into a formal language understandable to the Turing machine (the binary language, 1 and 0, so-called "machine language") through a compiler. Nonetheless, the Turing machine *does not solve that problem or implement that solution*. It only performs a series of logical operations. It does not interact with its environment. It cannot independently solve a concrete human problem in the same way that the determining judgment cannot be used to explain the living organism. The Turing machine can solve that problem *only if* the latter is interpreted and translated in a certain language and rules, i.e., in a certain computational architecture.[8]

[8] Computable functions are precisely those computable by lambda terms or general recursive functions. Alonzo Church and Alan Turing published independent papers that purported to demonstrate a general solution to the *Entscheidungsproblem*. A

Turner (2018: 67) recognizes this point:

> Without programming languages, machines would be idle devices much like cars without drivers or hairdressers without combs. […] General problem solving in these [machine] languages is difficult and unnatural because control is limited to instructions for moving data in and out of store, and representation is performed using numbers or Boolean values. These languages are for machines not humans.

Instead, high-level languages,

> employ more abstract concepts and control features such as procedures, abstract types, functions, polymorphism, relations, objects, classes, modules, and nondeterminism. These concepts aid problem solving and enable a more natural representation of the problem domain. They operate at a distance from the physical machine, and do not depend upon the architecture of any specific machine. This is made possible by layers of translation and interpretation (67).

This is the essential function of software: allowing the Turing machine to interact with the environment and solve concrete problems. Just as the faculty of judgment exerts a function of mediation between nature and understanding, software also mediates between

good number of solutions were proposed that all turned out to be extensionally equivalent. Obviously, this is not the place to deal comprehensively with computability: I refer manly to Turing (1936), Adams (1983), Copeland et al. (2013), Immerman (2011), Boolos et al. (2007). Regarding the analogy proposed in this paper, I consider computability as an abstract mathematical structure and software as a way of representing and interpreting this structure. I compare computability to the Kantian understanding: it is a set of mechanical laws that govern our way of thinking about the world.

concrete reality and the Turing machine. This is the analogy that I propose to develop in the following pages. The art of programming is the ability to see how the Turing machine can solve a specific concrete problem. All the different parts of programming (HLL, compilers, interpreters, specifications, etc.) are only different ways of bringing the problem and the Turing machine as close as possible.

In other words, the programmer performs the three fundamental operations indicated by Kant in *Logik* (§6): comparison, reflection, and abstraction. She/he compares the concrete problem and the Turing machine (let us call it its "computational resources").[9] After identifying the possible connections between the problem and the "computational resources", she/he reflects on these connections and elaborates her/his solution to the problem through the "computational resources" at her/his disposal. This solution is expressed through a formal architecture and a physical machine that implements this architecture. This solution has an analogical function because it allows us to connect the problem and the "computational resources", the concrete, and the abstract. This happens in two ways: a) *input* – software allows us to interpret the concrete problem in computational terms and in a language that can be understood by the Turing machine; b) *output* – software allows us to interpret the physical electrical impulses produced by the CPU as b.1) the solution of the problem or b.2) the physical expressions of the Turing machine abstract operations. Software reflection allows us to think that the Turing machine "solves" that problem, even if this is not the case, because the Turing machine does not "see" that problem properly. The problem solved by the Turing machine is not "our" problem (print a paper, booking a hotel, read the newspaper, etc.) but a series of mathematical functions.

[9] I use the expression "computational resources" here because there are many ways to understand computation and many ways to implement it.

Software mediation/reflection allows us to think that an abstract mathematical structure can implement physical operations.

5. The Software Imagination

There are two possible objections to the analogy between software and Kantian reflective judgment that I try to draw. The first is the most immediate: "Software does not have the form of a judgment, of a proposition, *then* we cannot compare software and reflective judgment". Obviously, programming languages are not formulated in natural language and, therefore, are not based on the same structures as natural language (subject + predicate, as Kant thinks). Programming languages are formal and based on a precise syntax or grammar. However, if we consider the Kantian notion of judgment in more general terms, as a power of synthesis expressed in a certain language, then it is possible to reply to the objection. Basically, programs are acts of synthesis between different *components* in interaction, thanks to *connectors*, in order to form a *system*. Then, this formula

$$x := x + 1;\ y := x * y$$

is tantamount to a proposition or a set of propositions. We can translate strings of code in propositions, or a set of propositions, and vice versa. This is the reason why programming languages can be understood by humans. [10]

The second possible objection is that our thesis reproduces, even if in alternative terms, those of Colburn and Turner: the programmer creates the harmony between physical and symbolic, between abstract and concrete, as a sort of *deus ex machina*. This is an

[10] The architecture of programs is determined, above all, by the paradigm that the programmer decides to follow. It can be *imperative* (or *procedural*), *functional*, *logic*, or *object-oriented*, each of which is connected to a precise conception of the program and of computation. Nevertheless, many languages can be mixed (Turner 2018: 67–76).

important objection because it gives me the opportunity to clarify a crucial point. The analogy/mediation of software does not come from a *creatio ex nihilo* in the programmer's mind.

Kant can again provide us with an important suggestion: at the roots of the three aforementioned operations (comparison, reflection, and abstraction) there is the faculty of transcendental imagination. The transcendental imagination produces the synthesis between the singular and the universal in determining judgment (transcendental schematism). It is always the transcendental imagination that produces the universal from the individual in the reflective judgment. For Kant, the transcendental imagination has a synthetic function that precedes and determines judgment and its logical forms.

I do not want to analyze the Kantian doctrine of imagination, which is not the subject of this paper (see Heidegger 1990; Sellars 1978). I want to formulate another question, which extends the comparison between software and reflective judgment that I try to formulate: how does the imagination work in software?

The programmer tackles a concrete problem and fixes requirements. Her/his job is to create a formal representation of this problem and these requirements. This is an act of imagination: the programmer creates an interpretation of the problem, which can be understood by the Turing machine. Why should this act be one of imagination? It is because the Turing machine and concrete reality cannot communicate—the Turing machine cannot understand the problem. Therefore, the programmer has to interpret the problem and create a new representation of this problem (the program) that can mediate between the problem itself and the Turing machine. This is an act of imagination.

Thus, in Kantian terms, the scope of DRJ is to reach a "free agreement" between problem, imagination (the program), and understanding (the Turing machine). This "free agreement" enables us to “see” the Turing machine as it solves that problem. The imagination enables the programmer to “see” how the Turing machine can solve the problem, even if the Turing machine cannot “see” the problem. Moreover, this agreement has to be effective. It requires implementation.

Given this, the imaginative act of software cannot be a *creatio ex nihilo*. It first has to be technical. This act must respect a series of technical constraints: rules, parameters, materials, etc. Furthermore, it has to a) express a language, b) have the ability to have a physical effect, i.e., to realize a causal action on the underlying material reality (the hardware, the instrument, the surrounding environment, etc.).

My hypothesis is that the imaginative act underlying software is realized through writing. Why do I choose writing? For two reasons.

First, software is writing; it is based on writing. For software, to be written is not a secondary property; it is its condition of possibility. Software would not be software if it were not written. Software is a form of writing that is not intended to be read as such; in fact, “for a computer, to read is to write elsewhere” (Chun 2013: 91). “Software is a special kind of text, and the production of software is a special kind of writing” (Sack 2019: 35).

Second, writing is a specific form of synthesis between the abstract and concrete. This is in two different senses:

- Writing is a synthesis between a language (abstract structures: grammar, syntax, semantics) and a material support (paper, clay, screen, etc.);
- Writing is a synthesis between language and space because it “spatializes” language, and this spatialization allows the visualization of language and, therefore, a completely new perceptive experience of the language. Spatialization and visualiza-

tion allow the discovery and invention of new uses of language and new concepts.

Writing is the name of a certain type of reason. As Bachimont (1996: 7) says:

> […] writing creates a spatial synopsis, allowing to identify relations and properties that remain undetectable in the linear succession of the temporality of the speech; writing shows relations which are not perceptible in orality. Indeed, by producing a spatial two-dimensionality of the content of the speech, mind can simultaneously access different parts of the content independently of the order connecting these parts in the oral flow.

Stressing the centrality of writing in knowledge, Bachimont (2001) speaks of a "graphical reason", i.e., a condition of possibility of "computational reason". In doing so, Bachimont extends the results of Goody (1977), the anthropologist who most contributed to understanding the role of writing in the emergence of certain cognitive operations or ways of thinking. In his famous work *The Domestication of the Savage Mind*, Goody (1977) shows the differential reasoning between written cultures and so-called oral cultures. Indeed, rather than attributing these differences to an axis of continuous progress on which oral cultures occupy a position inferior to written cultures, the former being an earlier state of the second, Goody shows that there are significant differences in thinking of the worlds between oral and written cultures and that these differences derive from the absence/presence of writing. Goody claims that writing has revolutionized human thought by producing autonomous conceptual structures and a specific relationship with the world. Writing introduces three main conceptual structures: the list, the formula, and the table. They form what we call "the graphical reason".

I now propose a combination of Bachimont's (2001) and Goody's (1977) reflections with those of the French designer and graphical artist Bertin (1983). In doing so, I distinguish three levels:

- Diagrammatical reason[11]
- Graphical reason
- Computational reason

I analyze only the last two levels and distinguish six conceptual structures (types of spatial synopsis) to which writing gives rise. I summarize them in this table. The structures of computational reason are a derivative of those of graphical reason. The first ones were described by Goody (1977) and the second by Bachimont (2001) (I changed his vocabulary a bit).

Graphical reason	LIST	FORMULA	TABLE
Computational reason	STACK	STRING OF CODE	NET

Each of these six structures is the expression of a synthesis between a language (a syntax, an alphabet, and a set of rules) and space. Each type of programming language comes from the interaction between the three conceptual structures of "computational reason". While the three structures in graphical reason (list, formula, and table) can work separately, in computational reason, they cannot: they must interact.

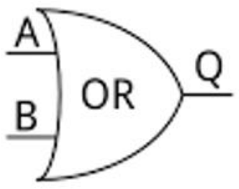
The stack is the basic form of data architecture: it allows the

[11] I use "diagrammatical reason" because I consider the writing as an evolution of the diagram. This is a thesis I am working on in another paper, which I cannot deal with extensively in this paper. As for the notion of diagram, I refer mainly to Peirce (1992, 1998), Stjernfelt (2007), and Bender and Marrinan (2010).

classification and spatial organization of data. The net allows communication between different stacks and combines stacks in a coherent whole. The string of code activates the relationships between the objects in the stacks. For instance, in Java programming language, the stacks correspond to classes, which include attributes and methods. Each string of code combines a method and an object and, therefore, makes the stacks interact through the net. The same thing can be said for other types of programming languages.

My claim is that any syntax and semantics of software language presuppose these conceptual structures. When the programmer uses terms such as "object", "operation", etc., she/he constructs their meaning through these syntheses of language and space.

The programmer could not draw this



or write this

```
Car
make:String, model:String, engine:Num
Find Speed, Start, Stop.
```

if she/he did not previously have the structures of diagrammatical and graphical reason and, then, those of computational reason. The art of programming is first an expression of diagrammatical and graphical reason. In creating her/his tools, the programmer writes formulas and draws schemes—she/he acts like a mathematician or logician. However, she/he has to overcome the diagrammatical and graphical levels of reason. The program is an evolution of diagrammatical and graphical reason.

The structures of computational reason are placed between the concrete problem and the Turing machine. In other words, they mediate the relationship between "understanding" in Kantian terms (the Turing machine) and "life" (the concrete problem to solve, the implementations, etc.). The functional and physical levels communicate, thanks to the mediation of these spatial syntheses. Moreover, thanks to its physicality, like writing, software can causally act on the physical machine (circuits) and have an effect in the world. I am not saying that the synthesis of the functional and physical levels *is* writing, only that *it is realized though* the spatial syntheses made possible by writing. It is thanks to the hybrid (conceptual and material) nature of these three spatial syntheses that we can "see" the functional level in the physical, the series of mathematical operations in the electrical impulses produced by the CPU, and, therefore, the Turing machine solving that problem.

6. The Software Delight

In the first book of the *Critique of Judgement* (§1), Kant (2016: 65) writes,

> If we wish to discern whether anything is beautiful or not, we do not refer the representation of it to the object by means of understanding with a view to cognition, but by means of the imagination (acting perhaps in conjunction with understanding) we refer the representation to the subject and its feeling of pleasure or displeasure. The judgement of taste, therefore, is not a cognitive judgement, and so not logical, but is aesthetic, which means that it is one whose determining ground cannot be other than subjective.

The judgement of taste is based on a certain agreement between imagination and understanding. This means that the judgement of taste is based on the subject's feelings, i.e., the manner in which the subject is affected by representations. The judgment of taste applies the principle of finality to the delight coming from the agreement between imagination and understanding, which Kant calls a "free play" of faculties. This delight is independent of any interest (see §2). In this delight caused by representations, the faculty of judgement finds the mark of finality. As Kant (2016: 67) says,

This relation, present when an object is characterized as beautiful, is coupled with the feeling of pleasure. This delight is by the judgement of taste pronounced valid for everyone; hence an agreeableness attending the representation is just as incapable of containing the determining ground of the judgement as the representation of the perfection of the object or the concept of the good. We are thus left with the subjective finality in the representation of an object, exclusive of any end (objective or subjective) consequently the bare form of finality in the representation whereby an object is given to us, so far as we are conscious of it as that which is alone capable of constituting the delight which, apart from any concept, we estimate as universally communicable, and so of forming the determining ground of the judgement of taste.

In DRJ, the agreement between imagination and understanding, which is realized through writing, delights the programmer. The programmer enjoys when the machine runs well and fast and solves the problem. This delight is seen as the expression of a finality: the machine acts for us and improves our world. Nevertheless, this finality is purely subjective. The machine is built by humans and responds to human purposes. Software specifications meet human criteria.

I see here an interesting parallelism between DRJ and the Kantian judgement of taste. In both cases, the agreement between imagination and understanding generates a delight that is the expression of the principle of finality. As we said earlier, the fundamental task of a programmer is to translate a problem (the singular concrete case) into computational terms (the understanding). She/he can do this only by using her/his imagination because there is no connection between concrete reality and the Turing machine. The programmer has to use her/his imagination. She/he creates the universal by which to think about the single individual case and make it comprehensible through understanding (the Turing machine). In doing so, she/he uses the imaginative structure of writing, the form of spatialization, and the materialization of language. The programmer reaches her/his scope only when she/he reaches an agreement between the concrete problem, her/his imagination, and the Turing machine. The program mirrors this agreement, which in turn makes the physical machine work and produces a delight. Through this delight, DRJ applies the principle of finality.

In the judgment of taste, the principle of finality is not the result of the operation of judging; it is its condition of possibility, namely, what guides the power of judging and makes it applicable. The same can be said for software. In her/his imaginative work, the programmer is oriented and guided by the principle of finality. This principle precedes and determines the syntheses of writing between the physical and functional levels. It precedes and determines all the stages of the programming.

The main way in which the principle of finality appears in programming is the act of design. Programming is essentially a design act: "design is everywhere in computer science" (Turner 2018: 128). The choice of what language to use and the algorithmic style is strictly connected to design choices and is, therefore, aimed at the construction of well-designed programs. Design is not just about beauty. "Design is a practice of creation turned towards the future and supported by an innovative intention" (Vial 2010: 44).

According to Turner (2018: 161), the hallmarks of a good digital design are 1) simplicity, 2) expressive power, and 3) security. However, Turner subordinates design to semantics: "More explicitly, the things that we may refer to and manipulate, and the processes we may call upon to control them, need to be settled before any actual syntax is defined. This is the 'semantics-first principle,' according to which, one does not design a language, and then proceed to its semantic definition as a post hoc endeavor; semantics must guide design" (169).

Even on this point, Turner's analysis appears to be characterized by an excessive intellectualization. As the French philosopher Stéphane Vial (2010) suggests, the objects of design are objects that have been submitted to a process of design, which consists of conceiving and producing effects that point to "experiences to be lived by means of forms" (115). The effects of design operate on the level of form, social meaning, and experience; thus, Vial sees design primarily as a "generator of human existence that proposes possible experiences" (65). In Vial's view, design deals not so much with the being as with events, not so much with the existing as with the new that will emerge. As the French designer Alain Findeli (2010) writes, the purpose of design is to improve the *habitabilité du monde*, i.e., our ability to live on this planet. Thus, design has a phenomenological and existential function, the aim of which is to improve the interaction between machines and humans and, therefore, between humans and the world. In this sense, design can be considered an extension of the use of Kant's principle of finality in the judgment of taste. The agreement between imagination and understanding produces a delight that is the expression of a finality in nature. Programmers' work makes the interaction between humans and machines possible through design as an expression of the principle of finality. Design is "a means for human beings to envision and realize new possibilities of creating meaning and experience and for giving shape and structure to the world through material forms and immaterial effects" (Folkmann 2013: 45).

From this point of view, I claim that design is the condition of programs, not the opposite. The programmer does not decide abstractly which objects to take into consideration: she/he deals with problems and has to choose the best strategy to solve them and give them meaning. The design criteria of elegance, correctness, simplicity, uniformity, modularity (the process of breaking up complex problems into smaller, simpler ones.), transparency, reliability, etc., shape the semantic and syntactic of the program. All possible criteria of the correctness of software are thought by the programmer through the principle of finality, i.e., through design. Writing is the first means by which software design is achieved.

I think that this view is closer to the way in which programmers understand their work. "One of the main reasons most computer software is so abysmal is that it's not designed at all, but merely engineered. Another reason is that implementors often place more emphasis on a program's internal construction than on its external design, despite the fact that as much as 75 per cent of the code in a modern program deals with the interface to the user" (Kapor 1996: 5). Moreover, software is not just a design job. It is also the source of a new form of design. "A discipline of software design must train its practitioners to be skilled observers of the domain of actions in which a particular community of people engage, so that the designers can produce software that assists people in performing those actions more effectively" (Denning and Dargan 1996: 112).

7. Conclusions

In this paper, I developed a series of criticisms of Turner's approach (2018) on software. In order to overcome the limits of Turner's approach, I proposed a definition of software from a transcendental Kantian perspective, i.e., through the concept of reflective judgment. I explain why and how we can consider software as a new form of reflective judgment, "digital reflective judgment". This judgement is realized through a type of imaginative synthesis that mediates between physical implementations and mathematical structures. I identified these structures as specific forms of writing that I called "graphical and computational reasons", following Goody (1977) and Bachimont (2001). Finally, I clarified my approach by showing the parallelism between software and the Kantian judgment of taste. In both cases, the principle of finality is an a priori condition.

I think that a Kantian approach to the question of software is a good model in explaining the nature of software in accordance with the concrete work of programmers. The transcendental approach to software avoids the main difficulties of Turner's approach outlined in section 2. In fact, I have shown that a transcendental approach is able to 1) explain the interaction between software and users through design; 2) escape the overly static framework of an ontology of the thing and then support an ontology of the process, which is much more suitable in explaining a phenomenon such as software; and 3) avoid an excessive intellectualization of software and highlight its creativity and the underlying work of the imagination.

References

Adams, R. (1983). *An Early History of Recursive Functions and Computability. From Gödel to Turing.* Boston: Docent Press.

Bachimont, B. (1996). *Signes formels at computation numérique*. http://www.utc.fr/~bachimon/Publications_attachments/Bachimont.pdf (accessed: October 2, 2020).

Bender, J., and M. Marrinan. (2010). *The Culture of Diagram*. Stanford: Stanford University Press.

Bertin J. (1983). *Semiology of Graphics*. Madison, WI: University of Wisconsin Press.

Boolos, G., J. Burgess, and Richard C. Jeffrey. (2007). *Computability and Logic.* Cambridge: Cambridge University Press (1st ed. 1974).

Callanan, J. (2008). Kant on Analogy. *British Journal for the History of Philosophy* 16 (4): 747–772.

Chun, W. (2013). *Programmed Visions. Software and Memory*. Cambridge: MIT Press.

Colburn, Timothy R. (1999. Software, Abstraction, and Ontology. *The Monist* 82: 3–19.

Copeland, Jack B., Carl J. Posy, and Oron Shagrir. eds. (2013). *Computability. Turing, Gödel Church, and Beyond.* London-Cambridge: MIT Press.

Denning, P., and P. Dargan. (1996). Action-centered Design. In *Bringing Design to Software*, edited by T. Winograd, 105–120. New York: ACM Press.

Eisler, R. (1994). *Kant Lexicon*. Translated by A.-D. Balmès and P. Osmo. Paris: Gallimard.

Findeli, A. (2010). Searching for Design Research Questions: Some Conceptual Clarifications. In *Questions, Hypotheses & Conjectures: Discussions on Projects by Early Stage and Senior Design Researchers*, edited by R. Chow, W. Jonas, G. Joost, pp. 23-36. London: Bloomington.

Folkmann, M. N. (2013). *The Aesthetics of Imagination in Design*. Cambridge: MIT Press.

Ginsborg, H. (2006). Empirical Concepts and the Content of

Experience. *European Journal of Philosophy* 14: 349–372.

Goody, J. (1977). *The Domestication of the Savage Mind.* Cambridge: Cambridge University Press.

Hanna, R. (2001). *Kant and the Foundations of Analytic Philosophy*. Oxford: Clarendon/Oxford University Press.

Hanna, R. (2005). Kant and Nonconceptual Content. *European Journal of Philosophy* 13: 247–290.

Hanna, R. (2006). *Rationality and Logic*, Cambridge, MA: MIT Press.

Hanna, R. (2017). Kant's Theory of Judgement. In *The Stanford Encyclopedia of Philosophy*, edited by E. Zalta. https://plato.stanford.edu/entries/kant-judgment/supplement4.html

Heidegger, M. (1990). *Kant and the Problem of Metaphysics*. Translated by R. Taft. Bloomington: Indiana University Press.

Ihde, D. (1990). *Technology and the Lifeworld*. Bloomington: Indiana University Press.

Immerman, Neil. (2011). Computability and Complexity. In *The Stanford Encyclopedia of Philosophy*, edited by E. Zalta. https://plato.stanford.edu/entries/computability/ (accessed: October 2, 2020).

Indurkhya, B. (2017). Some philosophical observations on the nature of software and their implications for requirement engineering. https://www.academia.edu/7817075/ (accessed: October 2, 2020).

Land, T. (2011). Kantian Conceptualism. In *Rethinking Epistemology, edited by* G. Abel et al., 197–239. Berlin: DeGruyter.

Land, T. (2015), Nonconceptualist Readings of Kant and the Transcendental Deduction. *Kantian Review* 20: 25–51.

Land, T. (2016), Moderate Conceptualism and Spatial Representation. In *Kantian Nonconceptualism,* edited by D. Schulting, pp. 145–170. London: Palgrave Macmillan.

Longuenesse, B. (1993). *Kant et le pouvoir de juger*. Paris: Puf.

Kant, I. (2016). *The Critique of Judgement*. Translated by J. Creed

Meredith. Scotts Valley: CreateSpace.
Kapor, M. (1996). A Software Design Manifesto. In *Bringing Design to Software*, edited by T. Winograd, 1–9. New York: ACM Press.
Kroes, P., and A. Meijers. (2002). The Dual Nature of Technical Artifacts – Presentation of a New Research Programme, in *Technè: Research in Philosophy and Technology*, 6: 23–46.
Peirce, C. S. (1992). *The Essential Peirce*, vol I. (1867–1893) (eds. N. Houser and C. Kloesel). Bloomington: Indiana University Press.
Peirce, C. S. (1998). *The Essential Peirce*, vol II. (1893–1913) (eds. N. Houser and C. Kloesel). Bloomington: Indiana University Press.
Sack, W. (2019). *The Software Arts*. Cambridge: MIT Press.
Sellars, W. (1978). The Role of the Imagination in Kant's Theory of Experience. In *Categories: A Colloquium*, edited by H. W. Johnstone Jr. Pennsylvania State University, 120–144.
Stjernfelt, F. 2007. *Diagrammatology*. Berlin: Springer.
Turing, A. M. 1936. On Computable Numbers, with an Application to the *Entscheidungsproblem*. *Proceedings of the London Mathematical Society*, *42*: 230–265.
Turner, R. 2018. *Computational Artifacts. Towards a Philosophy of Computer Science*. Berlin: Springer.
Vial, S. 2010. *Court traité du design*. Paris: Puf.