APPLIED METAPHYSICS – OBJECTS IN OBJECT-ORIENTED ONTOLOGY AND OBJECT-ORIENTED PROGRAMMING

By Gabriel Yoran

"As computer science works on domain-specific models in order to find solutions to practical problems, employing models of the world, informatics is – like any proper science – applied metaphysics."

Suggested citation: Yoran, Gabriel (2018). "Objects in Object-Oriented Ontology and Object-Oriented Programming." In *Interface Critique Journal Vol.1*. Eds. Florian Hadler, Alice Soiné, Daniel Irrgang
DOI: 10.11588/ic.2018.0.44744

ONTOLOGY AFTER INFORMATICS

"What can I know? What must I do? What may I hope for? What is man?"[1] The four Kantian questions, as universal as they seem, pivot around the I. All knowledge gained is knowledge only in the cognitive relation between acts of consciousness and an outside world, which is deemed more or less inaccessible. Every ethical demand is demanded of an I. Every hope experienced is experienced by an I. Kant holds that answering these three questions will inevitably lead to an answer of the fourth: What is man? And it is again an I who questions what it is. The Western world lives in the Kantian horizon. It pivots around the I.

Speculative realists set out to change that. While not representing a unified theory, this line of thought encompasses different non-anthropocentric positions striving to, in Ray Brassier's words, "re-interrogate or to open up a whole set of philosophical problems that were taken to have been definitively settled by Kant, certainly, at least, by those working within the continental tradition."[2] As overcoming the human as the epistemic center of the cosmos necessarily leads to both a speculative stance and a more or less realist position, speculative realism is a feasible term. In accordance with the tradition in which Kant named metaphysics "a wholly isolated speculative cognition of reason,"[3] speculative realism merely makes the nature of its task obvious by naming it accordingly.

The variant of speculative realism which will be looked into here, is object-oriented philosophy (more often referred to as object-oriented ontology and thus abbreviated OOO), a theory by contemporary American philosopher Graham Harman, who also coined the term. Even though OOO is subsumed under the speculative realism movement, Harman claims to be "the only realist in speculative realism."[4]

OOO, even though this is most likely unintended, is a substance ontology developed under the impression of informatics. It "might be termed the first computational medium-based philosophy, even if it is not fully reflexive of its own historical context in its self-understanding of the computation milieu in which it resides."[5] As "perhaps the first Internet or born-digital philosophy has certain overdetermined characteristics that reflect the medium within which [it has] emerged."[6] Such notions usually refer to the leading figures of speculative realism using blogs and social media to distribute their thoughts

1 Immanuel Kant, *Critique of Pure Reason*, ed. Paul Guyer and Allen W. Wood, The Cambridge Edition of the Works of Immanuel Kant (Cambridge: Cambridge University Press, 1998), A805/B833.
2 Ray Brassier, Iain Hamilton Grant, Graham Harman, and Quentin Meillassoux, "Speculative Realism," in *Collapse*, ed. Robin Mackay, vol. III (Oxford: Urbanomic, 2007), 308.
3 Kant, CPR, B xiv.
4 Graham Harman, personal communication with the author, March 12, 2017.
5 David M. Berry, Critical Theory and the Digital, *Critical Theory and Contemporary Society* (New York: Bloomsbury, 2014), 103.
6 Ibid., 104.

quickly and engage in lively discussions with the academic community online. OOO however has a deeper relation to the computational sphere: while Harman first publicly mentioned the term object-oriented philosophy in 1999,[7] object-oriented programming was already invented in the late 1960s – and the parallels between these two domains are noteworthy.

Working at the Norwegian Computing Center in Oslo, Ole-Johan Dahl und Kristen Nygaard in the 1960s conceived a new way of computer programming, in which what was separate before, namely data and functions, were molded into combined and somehow sealed logical units. Dahl and Nygaard named these units "objects" and the programming language they developed, Simula 67, is regarded the first to allow for software development following the paradigm of object-oriented programming (OOP).[8]

OOP has been in use for nearly five decades now and while it is still a popular way of structuring software development projects large and small today, its critics have become more vocal. OOP's unnecessary complexity is just one of the issues computer language designers bring up: "The problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle."[9] Regardless of OOP coming under fire lately, the striking parallels between the aesthetic and technological praxis of object-oriented programming on the one side and a new metaphysics on the other side, promise a fruitful contribution to the ontographic project.

As a science investigating "the structure and properties (not specific content) of scientific information, as well as the regularities of scientific information activity, its theory, history, methodology and organization," informatics was defined in the 1960s.[10] Since then the task of informatics has been extended beyond the analysis of scientific information and deepened by performing this task using the means of computing. Thus, informatics today has become the science that investigates the structure and properties of information. The similarities between object-oriented programming and object-oriented ontology do not come as a surprise, given that informatics is traditionally occupied with metaphysics: both computer science and philosophy "do not address the materiality of things such as physics, they are not confined to the 'science of quantity' (= mathematics)."[11] Since computer science strives to map reality onto computational structures,

7 Graham Harman, *Bells and Whistles: More Speculative Realism* (Winchester: Zero Books, 2013), 6.

8 Bjarne Stroustrup in: Federico Biancuzzi and Shane Warden, eds., *Masterminds of Programming* (Sebastopol, CA: O'Reilly, 2009), 10.

9 Joe Armstrong, *Coders at Work: Reflections on the Craft of Programming*, ed. Peter Seibel (New York: Apress, 2009), 213.

10 A.I. Mikhailov, A.I. Chernyl, and R.S. Gilyarevskii, "Informatika – Novoe Nazvanie Teorii Naučnoj Informacii," *Naučno Tehničeskaja Informacija*, no. 12 (1966): 35–39.

11 Alessandro Bellini, "Is Metaphysics Relevant to Computer Science?," *Mathema* (June 30, 2012), http://www.mathema.com/philosophy/metafisica/is-metaphysics-relevant-to-computer-science/.

employing substance ontologies seems obvious. As computer science works on domain-specific models in order to find solutions to practical problems, employing models of the world, informatics is – like any proper science – applied metaphysics.

PARALLELS

> Computational metaphors share a lot of similarity in object-oriented software to the principles expressed by [ooo's] speculations about objects as objects.[12]

There are astonishing parallels between object-oriented ontology and object-oriented programming, even though the former only borrowed the name from the latter.[13]

When object-oriented programming was invented, the dominant approach to computer programming was imperative or procedural. Imperative programming means conveying computational statements that directly alter the state of the program. A program designed in this way roughly works by linearly processing a list of functions step by step. When these statements are grouped into semantic units, "procedures," one can speak of procedural programming. Procedures are used to group commands in a computer program in order to make large programs more easily maintainable. Groups of statements also make code reusable, since the same set of statements can be invoked again and again. It also makes code more flexible, since parameters can be handed to a procedure for it to process. Parameters can be thought of as values handed to functions (the x in f(x)). While the function follows the same logics, the operation's result depends on the parameters passed.

These improvements however were not sufficient to handle complex computational tasks like weather forecasts. Tasks like this require simulations. And even though Alan Shapiro mockingly notes that "the commercialized culture of the USA is substantially not a real world anymore: it is already a simulation. Object-oriented programming is a simulation of the simulation,"[14] the necessity of simulating weather systems or financial markets called for more sophisticated strategies to structure computer programs. Instead of grouping lists of statements into procedures and have these statements directly manipulate a program's state, object-oriented programming offers a vicarious approach. Computational statements and data are being bundled together in objects. These objects are being closed off to the rest of the program and can only be accessed indirectly by means of defined interfaces. Under this new programming paradigm computer programmers became object designers – they were forced to

12 Berry, Critical Theory and the Digital, 205.

13 Graham Harman, personal communication with the author, August 18, 2013.

14 Alan Shapiro, *Die Software der Zukunft oder: das Modell geht der Realität voraus*, International Flusser Lectures (Cologne: König, 2014), 7; translation by the author.

come up with an object-oriented ontology for the world they wanted to map into the computer's memory.

The invention of object-orientation made object-oriented computer languages a necessity. The available computer languages did not possess the grammar necessary to describe objects and their relations. It becomes clear that "computer language" or "programming language" are misleading terms. These languages are products of human invention. They are human-designed, human-understandable languages, which computers can process in order to fulfill certain tasks. Designing a programming language is an attempt at producing the toolset for future developers to solve as yet unanticipated problems, sometimes in ways that were previously inconceivable. Object-oriented ontologies in informatics are pragmatic and open, they are realist in a sense of being a useful system of denotators of things outside the computer (or the programming language). They aim for reusable program code, which only needs to be written once, so problems do not need to be solved twice and errors do not have to be fixed in multiple places. Thus, the programming language designer's task is metapragmatic: designing a language as a tool for others to build tools to eventually fulfill certain tasks. Object-orientation discards lists of statements in favor of objects as the locus of, to use a Simondonian term, "problem solving." Simondon's notion of the individual describes objects as "agents of compatibilisation," solving problems between different "orders of magnitude."[15] With this notion Simondon seems to have anticipated the object in object-oriented programming; or at the very least, the actual implementation of objects in OOP prove to be in line with the traits of the individual Simondon described.

Object-oriented programming became so widely adopted partly because it is close to the everyday experience of objects. It also makes strong use of hierarchies, another everyday concept. Objects may remain identifiable and stable from the outside, even when their interior changes dramatically. The "open/closed principle" is evidence of this: a component, not necessarily an object, needs to be open for future enhancement, but closed with regards to its already exposed interfaces. This "being closed" ensures that other components depending on the component can rely on the component's functionality displayed earlier – unexpected changes in behavior need to be prevented.[16] Being closed can be read as unity, as a certain stability of an object that makes it identifiable. Object-oriented programming however reaches some of this stability by interweaving objects into a hierarchy, an idea that object-oriented ontology rejects.

In both object-oriented programming and object-oriented ontology objects are the dominant structural ele-

15 Gilbert Simondon, "The Genesis of the Individual," in *Incorporations*, ed. Jonathan Crary and Sanford Kwinter (New York: Zone, 1992), 301.

16 Bertrand Meyer, *Object-Oriented Software Construction*, Prentice-Hall International Series in Computer Science (New York: Prentice-Hall, 1988), 23.

ments. In object-oriented programming, objects are supposed to be modeled after real-life objects as the aim is to provide a sufficiently precise representation of the reality to be simulated. In practice this undertaking often fails. Objects are being created in code for things that do not exist outside the program. Functionality is forced into object form even when the result is awkward and unsatisfying. As a result, alternative programming paradigms are getting more interest lately and new programming languages like Apple's Swift are designed undogmatically, mixing different paradigms with the goal to always deliver the solution that's least error-prone for the use-case. But this should not be of any concern as we are focusing on the multitude of traits that OOP and OOO share:

1. Objects are both systems' basic building blocks.
2. Objects can be anything from very simple to extremely complex.
3. Objects have an inner life, which is not fully exposed to the outside.
4. Objects interact with other objects indirectly and do not exhaust other objects completely.
5. Objects can destroy other objects.
6. Results of interactions between objects may or may not be predictable from outside an object.
7. Objects can contain objects.
8. Objects can change over time, but at the same time stay the same object in the sense of an identifiable entity.
9. No two objects are the same.

OBJECTS AS UNPREDICTABLE BUNDLES

The first programming language regarded as object-oriented was Simula 67, invented in the 1960s by Ole-Johan Dahl und Kristen Nygaard at the Norwegian Computing Center in Oslo. Simula 67 was designed as a formal language to describe systems with the goal of simulation (thus the name Simula, a composite of simulation and language). Simula already incorporated most major concepts of object-orientation. Most importantly, Dahl's and Nygaard's object definition still holds today: objects in object-oriented programming are bundles of properties (data) and code (behavior, logics, functions, methods). These objects expose a defined set of interfaces, which does not reveal the totality of the object's capabilities and controls the flow of information in and out of the object. These two specifics are subsumed under the "encapsulation" moniker.[17]

Objects in programming are another variant of "the ancient problem of the one and the many":[18] they exist as abstract definitions, called

17 Biancuzzi and Warden, *Masterminds of Programming*, 350.

18 Graham Harman, *The Quadruple Object* (Winchester: Zero Books, 2011), 69.

"classes" or "object types," and as actual entities, called "objects" or "instances." So, while a class is the Platonic description of an abstract object's properties and behavior, instances are the actual realization of such classes in a computer's memory.[19] There can be more than one instance of any class, and it is possible and common for multiple instances of the same class to communicate with each other.

Let us look at a concrete example of the difference between procedural and object-oriented programming. In procedural programming, a typical function would be y=f(x), where f is the function performed on x and the function's result would be stored (returned) in the variable y. In object-orientation however, an object x would be introduced, which would contain a method f. An interface would be defined that would allow for other objects to call f, using a specified pattern. And so, by invoking f, the member function being part of object x – or x.f() for short – the object, containing both data and functionality, stays within itself. In our case, there is no return value, so no y to save the results of function f to. This is not necessary as the object itself holds all the data it operates on.

Object-oriented programming has been criticized for the fact that the behavior of object methods (functions inside objects) is unpredictable when viewed from a strictly mathematical perspective. A mathematical function y=f(x) is supposed only to work on x and return the result in y. An object method however can also modify other variables inside its object and thus lead to unpredictable results. A function is supposed to return its result – an object method however modifies its object, but does not necessarily return a copy of (or a pointer to) the whole modified object. When manipulating an object through one of its member functions, it is not known from the outside which effects this manipulation will have on the object internally. This means the object's behavior following such a method call is not predictable from outside of the object. While software developers generally try to prevent unpredictability, the object-oriented philosopher will hardly be surprised: it is a key characteristic of OOO that objects can behave in unpredictable ways and that their interiority is sealed off from any direct access:

> I think the biggest problem typically with object-oriented programming is that people do their object-oriented programming in a very imperative manner where objects encapsulate mutable state and you call methods or send messages to objects that cause them to modify themselves unbeknownst to other people that are referencing these objects. Now you end up with side effects that surprise you that you can't analyze.[20]

19 Vlad Tarko, "The Metaphysics of Object Oriented Programming," May 28, 2006, http://news.softpedia.com/news/The-Metaphysics-of-Object-Oriented-Programming-24906.shtml.

20 Biancuzzi and Warden, *Masterminds of Programming*, 315.

While in object-orientation data and operations performed on it need to be bundled into one object, the competing paradigm of functional programming means that operations and data are separated. In the functional programming language Haskell for example, functions can only return values, but cannot change the state of a program (as is the case in object-orientation).

THE PLATONIC CLASS

While objects may have complex inner workings (code as well as data), they usually do not share all this information with other objects. An object exposes certain well-defined interfaces through which communication is possible. In line with object-orientation's original application, we want to discuss the key concepts of OOP using a simulation program. We will imagine a program simulating gravitational effects in our solar system. Such a program, if designed in an object-oriented way, would most definitely contain an object type – or Platonic "class" – representing a planet. Such a class would contain variables to describe a planet's physical and chemical properties like its diameter, atmosphere, age, current average temperature, its position in relation to the solar system's sun, etc. It would also contain methods, which would be used to manipulate class data. A method to change the average temperature (to account for the case of a slowly dying sun for example) would need to be implemented as well. In a solar system simulation, there would be multiple instances – objects – of the planet class; in the case of our solar system one would create objects for Earth, Jupiter, Saturn etc.

The simulation would manipulate any planet's data by calling the object's respective method, for example the one to change the planet's average temperature on the surface. The actual variable holding the average temperature itself would not be exposed to the object's outside. So, any interaction with the object must be mediated through the interface methods provided by the object. All interactions with an object become structured by this intermediate layer and can be checked for faulty inputs. Instead of directly changing the temperature on a planet to a value below absolute zero (which would be possible if direct access was given), the intermediate data setting method provides its own logic, and thus limitations, to prevent such a "misuse" of the object.

But all planets are different and to take this into consideration in our simulation, we would need to set any instance's properties (data) accordingly. To do so, classes provide special "constructor" methods, which bring an instance of a class into existence. Constructors take parameters needed to initially construct an object and then create an instance accordingly. (To destroy objects, so-called "destructors" can be used as well.)

As mentioned, object-oriented programming differentiates between classes (object types) and objects (there is other terminology, but in this

work, we will use these classic terms as defined in the C++ programming language). What makes this parallel interesting is that it is an interplay between a fixed structure and free-floating accidents that constitutes an object. This interplay is what OOO deems an object's essence. As not to stretch the analogies between OOO and OOP too far, this interplay takes place on the inside of an object in OOO, but in OOP it crosses borders between objects. But similar to the situation in OOO, objects can come into existence without actively enacting any reality. However, the object structure in OOP (which we would call the counterpart to OOO's real-object-pole) defines what an object can do. This is to be understood as a potential and not as an exhaustive description of the object's capabilities. In OOP, the instance of an object (what we have come to see as its real-qualities-pole) cannot be reduced to the object itself (the real-object-pole) – an object therefore is always more than its rigid structure. If the object has any interface to the outside, which is the case with most objects in OOP, there is still no way to know the results of all possible interactions with the object.

HIERARCHY AND INHERITANCE

Let us assume all planets in our solar system simulation have been sufficiently defined. We would still need an object representing the sun. The sun is not a planet, but a star, yet there are properties and probably methods both share, something all celestial bodies incorporate. Since its first incarnation in Simula 67, using the object-oriented programming paradigm is synonymous with organizing objects hierarchically in tree-like structures. Every object has at least one parent object (a superclass) and can have child objects (subclasses). An object then inherits all properties and methods of its superclass (or, in some cases, superclasses) and hands them and its own properties and methods down its subclasses, which can then add additional properties and methods. So, both classes representing planets and suns should be derived from a superclass representing any celestial body. This celestial body class would then handle properties and methods shared by all its subclasses. Only methods and data necessary for more specific celestial bodies like planets or stars would be defined in their respective subclasses. In OOP, a principle of reversed subsidiarity is at work: anything that can be handled at the highest, most abstract level is being handled there; only more specific tasks are being handled further down the object hierarchy.

OOP's terminology, talking of "parent classes," "child classes," and "inheritance," shows the hierarchical tradition in which OOP is rooted. Any object in the hierarchy "inherits" all traits from its parent object. Such a hierarchy has at its root an abstract object (CObject in Microsoft's MFC model), which only consists of abstract methods that make no statement about the specifics of this object at all. Such an object is rarely being used directly by software developers,

but only through one of its more concrete subclasses. But not all objects are part of such a hierarchy, like for example the CTime object in the MFC model.[21] CTime is used to represent an absolute time value. Operations on such a value are very basic and needed in a multitude of methods, but it would be hard to logically position a time object somewhere in an all-encompassing hierarchical system. The question of what a representation of a specific time should be derived from is hard to answer. This concept is too basic to be inserted into a hierarchy. So, while CTime objects can be integrated into custom-made hierarchies, they themselves are not derived from any superclass: representations of time are solitary objects within the MFC model.

INTERFACE AND IMPLEMENTATION

Now that we have a small hierarchy of celestial bodies represented in our object-oriented program design, we still face the task of implementing the actual simulation algorithm. Discussing this algorithm itself is outside our scope. We are more interested in where such an algorithm would be placed in an object-oriented design. This touches a key question of any object-oriented system: where and how do processes take place? Do they happen within objects, between objects, or in both places? While Simondon stresses the notion of objects as being through becoming,[22] the concepts of both OOP and OOO define objects qua their relative stability.

In object-oriented ontology, real objects need sensual objects as a bridge between them, leading to a chain of objects. Sensual or real objects cannot touch each other directly. The sensual object acts as an interface between real objects – or the real object as the interface between sensual objects. In object-oriented programming, objects cannot touch directly as well: they are broken down in interface and implementation parts. The interface part acts as an – incomplete – directory of methods and variables made available to other objects. It never exposes everything on an object's inside to the outside. It can even announce methods, which at the time of such an announcement are not even fully defined. Only when these methods are being invoked, a real-time decision will be made in regard to which version of the method would be appropriate to use in the current situation. So, OOP's interface is on the one hand a sensual object since it serves as the interface to other objects while not exposing the whole enactability on reality of its real object – which would be the implementation. Methods can execute different code, depending on criteria inaccessible from the outside, allowing for a program to change during runtime without damaging the

21 Microsoft, "CTime Class," 2015, https://msdn.microsoft.com/en-us/library/78zb0ese.aspx.

22 Simondon, "The Genesis of the Individual."

object's identifiability. The implementation part on the other hand represents the real object in the totality of its enactability in the program.

As for the solar system simulation, in object-oriented programming the obvious implementation would be a superclass representing all the components of a solar system needed for its simulation on a celestial bodies' level. An instance of such a solar system class would then have to incorporate member classes for every celestial body in the solar system. But which object would be the one to describe the relations between all the data and methods of the solar system object? One could create methods in the solar system class that would contain the algorithm needed for the simulation, like modifying a planet's position in space depending on the position and movement of other celestial bodies as time progresses. But the intended way of handling such a simulation is a technique called message-passing.

Objects can send and receive messages. The concept of message-passing allows for messages to be sent to an object, which then decides how to handle the message. This way an object is able to handle requests dynamically, depending on the type of data sent to it. This illustrates how both sides in an object-to-object interaction are involved. This interaction is not a simple sender-receiver relationship, but a rich exchange in which both objects involved do not fully touch each other, but are selective with regards to which input to accept at all. An object representing a planet could send a message to other planet objects, informing them about its own location in space. These other planets then would change their position in space accordingly. This way one could create a very simple simulation of gravity, but none of the objects involved would have any access to other object properties not needed for the calculation of gravitational effects.

So, message-passing is not just a concept of inexhaustibility, it is also a concept of indirection. Objects do not exhaust each other, they do not even touch directly, but they communicate by messages, which can be seen as an implementation of the concept of sensual objects.

INEXHAUSTIBILITY OF PROGRAMS

Let us go back to the solar system simulation example one last time. We found that the object ontology offered by object-oriented programming languages is a lax one, since there can be objects outside the hierarchy.

The solar system object, the object which hosts our simulation, would need to be instantiated at some point, since it cannot create itself. There has to be code outside the solar system class. Of course, there might be another object, which again incorporates the solar system class (a superclass to the solar system) representing a galaxy. But the Milky Way is not useful for simulating the gravitational effects in our solar system, and this would just move the problem to another level. The object-oriented programming paradigm is

an *abstraction* from the hardware the program will eventually be running on, since the central processing unit (CPU) does not "know" objects. The compiler or interpreter program must have done its task of translation to machine code before the CPU can run the program – and after this translation the object concept is lost to the CPU. These translator programs reduce object-orientation to a very basic sequence of memory operations, which the chip can process. This would only change if object-oriented hardware were being built, hardware that would render compilers or interpreters useless – but object-oriented chip designs like the Intel iAPX 432, which was introduced in 1981, eventually failed. They were slow and expensive and new technologies more suitable to the limitations of hardware prove more efficient – and so the idea of object-orientation in chips has only found very limited application.[23]

Programming languages came a long way in the last 60 years. They moved from a primitive set of commands in order to directly access a processor's memory to complex semantics, completely abstracted from the hardware its programs will run on. All high-level programming languages need an intermediary between statements made in such a language and the hardware programs are supposed to run on – these intermediaries are either compilers (programs that in a time-consuming way translate high-level programming languages to machine code the processor can work with) or interpreters (which basically fulfill the same task in real-time). In any case, there is a medium between the high-level language and the machine.[24]

While objects in object-oriented ontology are described as broken down in a real and a sensual part (what we superficially likened to the concepts of implementation and interface in programming), we need to understand that the whole relation of the statements made in a high-level programming language to the hardware the written program will run on is the relation of model and reality. The hardware of the chip forms the ultimate reality of the program, since the hardware defines the reality against the model put on top of it must work. The reality of the hardware again is its context, the wider environment of the machinery, its applications, and the people using it.

The limits of a program's enactability of its reality are in the hardware it runs on and the time available. A self-modifying program could enact an infinite amount of reality given there is enough time. So, the real object is inexhaustible by the relations it enters into with sensual objects. Programs running on a chip can never exhaust it. It is impossible to list all the programs that could be executed on the chip. It is not even possible to know in advance if all these programs will actually come to an end. Alan Turing described this phenomenon, which later became known as the

23 David R. Ditzel and David A. Patterson, "Retrospective on High-Level Language Computer Architecture"(ACM Press, 1980), 97-104, doi:10.1145/800053.801914.

24 A new generation of chips might end this separation. FPGAs are chips whose hardware can be modified by means of software, effectively blurring the line between software and hardware.

"halting problem": it is undecidable if an arbitrary computer program will eventually finish running or will continue running forever.[25] The halting problem extends inexhaustibility to the proof of inexhaustibility.

Object-oriented ontology aims at treating all objects equally – which rules out a central perpetrator. In object-oriented programming, it seems that there is no central perpetrator as well and objects act independently from a central instance. In reality, object-orientation today is a paradigm put on top of hardware, which is incapable of working without a central perpetrator. So, while the language in which the program is modeled, is object-oriented, it is important to understand that these objects are constructions in a language, which again tries to mimic things and relations in reality.

Objects act on behalf of themselves as long as one stays at the object's level of abstraction. On the chip's level these objects are nonexistent – the CPU only acts upon memory, where certain information is stored. The CPU and the operating system will make decisions without the objects "knowing," for example for dispatching: since programs today mostly run on computers with more than one central processing unit, it is necessary to distribute tasks (or object methods) to different CPUs.

The intuition of being surrounded by objects with a certain independence from each other is at the root of both models, OOP and OOO. But object-oriented ontology rejects the concept of a reducibility of objects to other objects: even though every object can be broken down to its parts (representing new objects): these objects do not exhaust the bigger object they form. There is nothing "below" objects in OOO. OOP however is a model, which is deliberately put on top of the more primitive and non-intuitive computational concept of memory.

This shows how object-oriented programming works only at a certain level of abstraction, thus constituting the major difference between object-oriented programming and object-oriented ontology: the earlier being a model applied pragmatically in one domain, the latter aiming for a complete metaphysics.

25 Alan M. Turing, "On Computable Numbers, with an Application to the Entscheidungsproblem," *Proceedings of the London Mathematical Society* s2-42, no. 1 (January 1, 1937): 230-65, doi:10.1112/plms/s2-42.1.230; Alan M. Turing, "On Computable Numbers, with an Application to the Entscheidungsproblem. A Correction," *Proceedings of the London Mathematical Society* s2-43, no. 6 (January 1, 1938): 544-46, doi:10.1112/plms/s2-43.6.544.

REFERENCES

Armstrong, Joe. Interview by Peter Seibel. In: *Coders at Work: Reflections on the Craft of Programming*, edited by Peter Seibel, 205-239. New York: Apress, 2009.

Bellini, Alessandro."Is Metaphysics Relevant to Computer Science?"Mathema (June 30, 2012). http://www.mathema.com/philosophy/metafisica/is-metaphysics-relevant-to-computer-science/.

Berry, David M. *Critical Theory and the Digital. Critical Theory and Contemporary Society*. New York: Bloomsbury, 2014.

Biancuzzi, Federico, and Shane Warden, eds. *Masterminds of Programming*. Sebastopol, CA: O'Reilly, 2009.

Brassier, Ray, Iain Hamilton Grant, Graham Harman, and Quentin Meillassoux. "Speculative Realism." In: *Collapse*, edited by Robin Mackay, III:306-449. Oxford: Urbanomic, 2007.

Ditzel, David R., and David A. Patterson. "Retrospective on High-Level Language Computer Architecture."In: *Proceedings of the 7th annual symposium on Computer Architecture*, 97-104. ACM, 1980. doi:10.1145/800053.801914.

Harman, Graham. Bells and Whistles: *More Speculative Realism*. Winchester: Zero Books, 2013.

Harman, Graham. *The Quadruple Object*. Winchester: Zero Books, 2011.

Kant, Immanuel. *Critique of Pure Reason*. Edited by Paul Guyer and Allen W. Wood. The Cambridge Edition of the Works of Immanuel Kant. Cambridge: Cambridge University Press, 1998.

Meyer, Bertrand. *Object-Oriented Software Construction*. Prentice-Hall International Series in Computer Science. New York: Prentice-Hall, 1988.

Microsoft. "CTime Class," 2015. https://msdn.microsoft.com/en-us/library/78zb0ese.aspx.

Mikhailov, A.I., A.I. Chernyl, and R.S. Gilyarevskii. "Informatika – Novoe Nazvanie Teorii Naučnoj Informacii." In: *Naučno Tehničeskaja Informacija*, no. 12 (1966): 35-39.

Shapiro, Alan. *Die Software der Zukunft oder: das Modell geht der Realität voraus*. International Flusser Lectures. Cologne: König, 2014.

Simondon, Gilbert. "The Genesis of the Individual." In: *Incorporations*, edited by Jonathan Crary and Sanford Kwinter, 297-319. New York: Zone, 1992.

Tarko, Vlad. "The Metaphysics of Object Oriented Programming." *Softpedia News* (May 28, 2006). http://news.softpedia.com/news/The-Metaphysics-of-Object-Oriented-Programming-24906.shtml.

Turing, Alan M."On Computable Numbers, with an Application to the Entscheidungsproblem." In: *Proceedings of the London Mathematical Society* s2-42, no. 1 (January 1, 1937): 230-65. doi:10.1112/plms/s2-42.1.230.

Turing, Alan M. "On Computable Numbers, with an Application to the Entscheidungsproblem. A Correction." In: *Proceedings of the London Mathematical Society* s2-43, no. 6 (January 1, 1938): 544–46. doi:10.1112/plms/s2-43.6.544.