The Invention of the Object: Object Orientation and the Philosophical Development of Programming Languages

Justin Joque

Abstract

Programming languages have developed significantly over the past century to provide complex models to think about and describe the world and processes of computation. Out of Alan Kay's Smalltalk and a number of earlier languages, object-oriented programming has emerged as a preeminent mode of writing and organizing programs. Tracing the history of object-oriented programming from its origins in Simula and Sketchpad through Smalltalk, particularly its philosophical and technical developments, offers unique insights into philosophical questions about objects, language, and our digital technologies. These early attempts to understand objects as basic elements of computation demonstrate the ways in which language, while firmly planted in the material reality of computation, must delimit objects from each other. This essay critically explores this history and explicates a theory of objects suggested by the development of object-oriented programming languages, which insists on the importance of language for representing and delimiting objects. It argues that the philosophies behind object-oriented programming are ultimately opposed to the claims of object-oriented ontology and find themselves more closely allied with philosophies that insist on the mediation of what exists through language.

Keywords

Programming languages
Object-oriented programming
Philosophy of computing
Object-oriented philosophy

This is a preprint version of the manuscript. The final publication will be published in *Philosophy and Technology and* available at Springer via http://dx.doi.org/10.1007/s13347-016-0223-5

1. The Invention of the Object

With the growing threat of ecological catastrophe, our increasing dependence on global material flows of goods and resources, and the realization that even our virtual worlds require massive material infrastructure, a number of diverse philosophical and scholarly discourses are turning (and returning) to considerations of materiality and explicitly the world of objects. Most notably, various forms of speculative realism (Mellasioux) and

object-oriented ontology (Harman; Levi-Bryant; Morton; Bogost; while not directly recognized as a proponent of this ontology, Latour should count at the very least as a fellow traveler or perhaps a predecessor) have insisted on the importance of objects, especially the relations between objects beyond our limited human view. Many of these philosophies—especially Harman's—are based on the notion of solid objects that exist autonomously from each other. Despite the obvious allusion to object-oriented programming in the naming of object-oriented ontology, there are few descriptions of the relationship between object-oriented programming (OOP) and said ontology. This is especially unfortunate as the history and philosophy that surround object-oriented programming offer a nuanced understanding of objects, their ability to hide part of themselves from the world, their relations, and their representation in languages that in many ways challenge the claims offered by object-oriented ontology. The philosophies that underlie OOP, likely as a result of the exigencies of creating functional systems, stress the relations between objects and the difficulties in conceptualizing objects as fully autonomous outside of the languages that address them.

In order to fully explicate the technical-philosophical ideas that have been developed under the name of object-oriented programming, it is imperative to trace the ways in which the concept of an "object" as an element of programming was both envisioned and forced to change as a result of the practical and theoretical demands of the actual work of programming. This history suggests that both objects themselves and even the concept of what an object is are unstable and emergent.[1] Developments in object-oriented programming continue, but perhaps, the most telling points of its philosophical implications are its earliest conception in Alan Kay's Smalltalk and the two major influences that preceded it: Simula, a language designed for simulating complex systems, and Sketchpad, a program for creating computer-assisted drawings. While histories and explanations exist for all three languages, the present essay focuses on the technical means by which the notion of an object was created and the philosophical paths that brought programming to this point.[2] As such, the main sources are largely drawn from the technical and philosophical descriptions laid out by the designers of these languages.[3] Furthermore, it should be noted that these sources present a rigorous definition of how to understand programming that is likely rarely to be held up in the messy practice of writing actual programs. The most eloquent notions of how a program should be written often fall away under the stress of deadlines and the collaborative work of committees. Moreover, even the definitions of objects and other programming concepts offered are often ideal and aspirational; in the negotiation of actual implementation, they are frequently further unsettled as they are implemented in actual systems.[4] Still in order to appreciate the relationship between objects and language, it is fruitful to follow the

development of the lofty ideas that outline this type of programming and the groundbreaking innovations that have led to it.

The term object-oriented programming describes both a style of writing computer code and a group of programming languages that are designed to be programmed in such a way. OOP offers unique insights into the creation of digital objects and the ways in which they are represented in a constantly developing field of languages. The advent of object-oriented programming exists within a much longer trend towards increasing abstraction in computer languages. This trend has moved programming languages further and further away from directly describing and operating on the individual bits, inputs, outputs, pixels, and switches that make up the hardware of a given computer to abstract concepts such as functions, objects, classes, and graphics. The history of programming languages runs from the use of addresses of small blocks of memory in a system (the actual physical storage of bits on hard drives and other media) to the use of primitive data types like integers, characters, and floats (variables with a floating decimal point); to more complex data types such a strings, lists, and arrays; and finally to complex, programmer-defined data structures that composite a variety of simpler data types. What is so striking about the work carried out in the 1960s and 1970s is precisely the way in which this increasing abstraction allowed programmers to think about "objects" as a useful metaphor for what was happening on the level of hardware or perhaps more accurately on the level of interaction between programmer and hardware. We can glimpse in this history a unique moment when a number of limited functional sets of operations proliferate, abstract, and coalesce to create a rich and diverse set of languages that begin to speak of objects, classes, and public and private properties.

This trend towards abstraction has not occluded the possibility and necessity of programming at lower levels. Programmers still write machine code, especially when computational efficiency is extremely important or one needs to interact with the most basic aspects of a machine (e.g., parts of the program that loads the operating system and code written for certain microcontrollers). This history of increasing abstraction is not one of direct succession but rather the diversification of a complex ecology of programming languages. Moreover, there have been and continue to be other frameworks for abstracting digital systems. Thus, the point is not to suggest that the object is somehow an inevitable and necessary outcome of this process of linguistic abstraction. Rather, this history demonstrates how the computational object, as one possible outcome for a process of abstraction, offers a number of critical insights into the relationship between objects, language, and the real.[5]

2. Object-Oriented Programming

The notion that a program can be thought of as a set of objects that consist of a series of properties and functions, which ultimately allow them to interact with other objects, provides the central metaphor for object-oriented programming. Each of these objects is defined by a class, which lays out its basic rules, and provides a method for constructing as many objects as needed. If this all seems rather abstract, it is because its entire purpose is, in the most general and abstract terms, to describe programming any system. Object-oriented programming provides an abstracted means to think about the real work that a program does. Many introductory programming books usually begin with a simple example to clarify the matter and perhaps one would help here as well.

We can take an example from the manuals for Simula, a system to manage (or simulate) an airport ticket counter. We could design our system with a class named "airplane" with a fixed number of seats and a function allowing a passenger to be assigned to a seat. We could also create a class named "passenger" that could perform various functions and have data assigned to it. The class itself does not contain the specific data and instead functions as a template from which to create objects. Our program then would take these classes, and every time it needed a new passenger or airplane, it would create an instance or object that would maintain this data as long as needed. This form of programming helps to conceptualize complex programs and also to divide the writing of such programs among multiple programmers as each programmer can focus on a certain class or set of classes once they are defined.

3. Language or Real Language

In order to appreciate the novelty and importance of the concepts developed under the name object-oriented programming, it is critical to understand how programming languages function not only as a means of instructing a computer but also a tool for thinking and communicating, in short as a language. Rather than being merely a means to instruct computers, they are complex languages whose logics and ambiguities continually reshape computation. There is a tendency both in general discussions and critical examinations of software to separate "language" from code. These arguments most often center on the claim that, because computer languages are compiled by a strict set of rules into machine code, they are merely a set of instructions for the computer to follow and cannot signify anything or provide any ambiguity like a real human language. Thacker (2004, pp. 13–14) presents this position succinctly, saying that "code is not necessarily language, and certainly not a sign…A code is a series of activated mechanical gears, or a stack of punched cards circulating through a tape-reading machine, or a flow of light pulses or bits in a transistor or on silicon." While Evens (2006, p. 89) ultimately makes an argument for creativity and desire in the interface between programmer and machine, he simultaneously suggests that

“computer language is wholly literal. Every line of code derives its meaning precisely from the letters or characters that are used to write it down, and it has no meaning beyond those letters.” Kittler (1995, p. 147) even insists that “the last historical act of writing may well have been the moment when, in the early seventies, Intel engineers laid out some dozen square meters of blueprint paper (64 square meters, in the case of the later 8086) in order to design the hardware architecture of their first integrated microprocessor.”

Ultimately, this antipathy to seeing programming languages as language and insistence on the importance of the computational translation of code into binary, which shows up in stronger and weaker forms in a variety of additional authors including Hayles (1999), Galloway (2004, 2006), and Golumbia (2009), finds an early expression in Derrida’s repeated distinction between writing and “the program.” For instance, in *Circumfessions* (1993, p. 31), Derrida says against an attempt to codify his theoretical edifice, “The grammar of his theologic program will not have been able to recognize, name, foresee, produce, predict, *unpredictable things* to survive him.” Here, the program is presented as an attempt to refuse the unpredictable that is endemic to writing. It is this belief that the program is closed and completely predictable in its translation from computer language to binary code that underwrites these repeated claims that computer language is wholly literal, a literality that can never be literature, and hence not language.

While computer languages function and develop differently than spoken or traditional written languages, they are still creative, metaphoric, and evolving systems of communication. Even though an effective program must be interpreted and compiled into binary code following a rigorous set of rules, the entire edifice of contemporary high-level computer programming languages has been built around making it easier for a programmer to implement his or her ideas and communicate that implementation with others who look at the code. Nearly, every contemporary programming language has a method for marking lines as comments so that a programmer can explain to others (and herself in the future) the purpose of parts of the program. Names of key parameters are decided in such a way as to make the flow of the program comprehensible, as the compiler is largely agnostic towards the selection of these names. Moreover, the challenge of writing complicated programs is often dividing the program into smaller subsets and designing a method to make sense of these divisions. I recall my high school programming textbook beginning with the statement that a good programmer can write code that works, but an excellent programmer can write code that another programmer can easily understand.

```
#include <stdio.h>

int main(void){
  printf("hello, world\n");
  return 0;
}
```

```
0000000 64207 65261 00007 00256 00003 32768 00002 00000
0000010 00016 00000 01296 00000 00133 00032 00000 00000
0000020 00025 00000 00072 00000 24415 16720 17735 17754
0000030 20306 00000 00000 00000 00000 00000 00000 00000
0000040 00000 00000 00001 00000 00000 00000 00000 00000
0000050 00000 00000 00000 00000 00000 00000 00000 00000
0000060 00000 00000 00000 00000 00025 00000 00552 00000
0000070 24415 17748 21592 00000 00000 00000 00000 00000
0000080 00000 00000 00001 00000 04096 00000 00000 00000
0000090 00000 00000 00000 00000 04096 00000 00000 00000
00000a0 00007 00000 00005 00000 00006 00000 00000 00000
00000b0 24415 25972 29816 00000 00000 00000 00000 00000
00000c0 24415 17748 21592 00000 00000 00000 00000 00000
00000d0 03856 00000 00001 00000 00045 00000 00000 00000
00000e0 03856 00000 00004 00000 00000 00000 00000 00000
00000f0 01024 32768 00000 00000 00000 00000 00000 00000
0000100 24415 29811 25205 00115 00000 00000 00000 00000
0000110 24415 17748 21592 00000 00000 00000 00000 00000
0000120 03902 00000 00001 00000 00006 00000 00000 00000
0000130 03902 00000 00001 00000 00000 00000 00000 00000
0000140 01032 32768 00000 00000 00006 00000 00000 00000
0000150 24415 29811 25205 26719 27749 25968 00114 00000
0000160 24415 17748 21592 00000 00000 00000 00000 00000
0000170 03908 00000 00001 00000 00026 00000 00000 00000
0000180 03908 00000 00002 00000 00000 00000 00000 00000
0000190 01024 32768 00000 00000 00000 00000 00000 00000
00001a0 24415 29539 29300 28265 00103 00000 00000 00000
00001b0 24415 17748 21592 00000 00000 00000 00000 00000
00001c0 03934 00000 00001 00000 00014 00000 00000 00000
00001d0 03934 00000 00000 00000 00000 00000 00000 00000
00001e0 00002 00000 00000 00000 00000 00000 00000 00000
00001f0 24415 28277 26999 25710 26975 26222 00111 00000
0000200 24415 17748 21592 00000 00000 00000 00000 00000
```

A simple program to print the words "hello, world" written in C (above) and the beginning of the same program compiled and displayed as hexadecimal (right) suggest the major advantages to programming in human readable languages over machine code

The entire purpose of high-level languages is to create human-comprehensible abstractions that allow more efficient programming by creating ways to describe computation that exist as a translation point between human abstractions and hardware. The first version of Simula was explicitly seen as both a tool for programming and communication. The language was initially conceptualized as a language for writing simulations; as such, it was designed to be both a compilable programming language to run simulations and a language to describe the real-world systems that were being simulated. The manual for Simula 1, the first version of the Simula family of languages, states, "Attempts have been made to make the language unifying—pointing out similarities and differences between systems, and directing—forcing the research worker to consider all relevant aspects of the systems. Efforts have also been made to make Simula descriptions easy to read and print and hence a useful tool for communication" (Nygaard and Dahl 1978, p. 249). Thus, even at this relatively early stage in the development of high-level programming languages and object-oriented methods, there was a deep concern with providing for human communication.

Ultimately, while the compilability of a program and its set of rules for translation into binary code constrain the possibilities of computer programs and language, the use of programs to communicate between humans and the metaphors and abstractions permitted by object-oriented languages produce languages that approximate the complexity and creativity of "real languages." To create effective means for programming requires developing a language and system to translate the "real" world into computation and interact with already-established modes of computation. To deny that programming languages are somehow less real, less creative or have less to tell us about the world than other languages

proscribes a powerful opportunity to interrogate a whole class of rapidly changing languages.

4. The Pre-History of the Object I: Simula

With these prefatory matters in mind, it is now possible to turn directly to the history of the object as a means of thinking and writing programs. The earliest computer programs were written in machine code, which operated directly on individual memory locations (each block of memory has a unique address that high-level languages hide from the programmer) and provided only the operations that the processor supported (e.g., add, multiply, and Boolean logic). By the end of the 1940s, computers began including assemblers, which would translate assembly code into machine code. Assembly languages provided the major advantage of mnemonics or programmer-defined names that could be used to refer to commands and memory locations rather than numbers used with machine code (Salomon 1993). The mid-1950s saw the development of higher-level languages that began to abstract some of the functions being done by the hardware into more human-understandable concepts. These languages began to manage memory without requiring the programmer to deal directly with locations in memory.

These early high-level languages were generally procedural; programs were constructed as a sequential set of commands that followed one after the other (and would at certain points return to an earlier point in the program or repeat a section or skip sections under certain conditions). Structured programming approaches were added to allow programmers to reuse parts of programs and minimize convoluted movements through the program creating what is known as “spaghetti code.” Out of these attempts, to more coherently structure programs, objects began to emerge. One of the earliest languages to begin moving away from procedures towards objects was Simula, developed in the 1960s by Ole-Johan Dahl and Kristen Nygaard at the Norwegian Computing Center (NCC).

Nygaard began the initial designs for Simula in 1960 upon his arrival at NCC. Simula went through a number of revisions and was released as two separate languages. The first functioning version, Simula I, was completed by the end of 1964 for a UNIVAC 1107 computer as part of an agreement between UNIVAC and NCC. Simula I was designed primarily as a language for simulating complex systems and processes. The initial design framework that Nygaard and Dahl began underwent significant revisions in the early 1960s before they produced a functional compiler. The original concept for the language was based around a series of stations, each of which would handle a queue of customers. The customers were treated as passive data structures; they only contained variables (in the case of an airport passport number, seat assignment, whether their ticket was paid, etc.). The

stations, as active processes, would then interact with the customers' variables and assign the customers to a different station's queue (or remove them from the program). Apparently, this early design was influenced by the desire to simulate an airport departure system, as it is consistently used as an example throughout Nygaard and Dahl's discussion of the framework (Nygaard and Dahl 1978; Krogdahl 2003). They attempted to find language to describe what exists in the world and at the same translate that language into computation. Even in its very early stages, we can see how intimately the development of object-oriented programming touched on philosophical and ontological questions.

```
system Airport Departure:= arrivals,counter,
     fee collector, control, lobby;
customer passenger (fee paid) [500] ; Boolean
          fee paid;
.......
station counter;
     begin accept (passenger) select:
          (first) if none: (exit);
     hold (normal(2,0.2));
     route (passenger) to:
     (if fee paid then control else fee collector)
     end;
station fee collector, etc. .....
```

An example SIMULA I code snippet demonstrating the basic structure for an airport departure system from "The Development of the Simula Languages."

As Nygaard and Dahl wrestled with these concepts prior to the implementation of Simula I, they began to abstract this dual ontology of a network of active stations and passive customers towards a single ontology of processes. Instead of representing the system as consisting of separate passive and activate processes, they began to shift towards a system of processes that contained both active and passive elements (Holmevik 1994). In a 1978 text written by Nygaard and Dahl describing the development of Simula, they explain

The airport departure system could be considered from a "dual" point of view: It could be described by active passengers, grabbing and holding the passive counter clerks, fee collectors etc. Then we realized that it was also possible to adopt an "in-between" or "balanced" point of view: describing the passengers (customers) as active in moving from station to station, passive in their interaction with the service parts of stations. These observations seemed to apply to a large number of situations. Finally, in our search for still wider classes of systems to be used to test our concepts, we found important examples of systems which we felt could not naturally be regarded as "networks" in the sense we had used the term…The result of this development was the abandonment of the "network" concept and the introduction of processes as the basic, unifying concept (p. 249).

We can witness here a very condensed process of (programming) linguistic abstraction. The initial language, based on a certain system, is confronted with different concrete systems that force the abstraction to change and account for these new systems. Thus, the dualistic system of stations and customers is replaced by a more nimble and robust metaphor of processes. Moreover, around the same time that they introduced the notion of processes, the language formally differentiated the description of a process from the process itself. Within Simula I, a process is declared with the keyword "activity"; thus, a program could be thought of as describing "activities," which then were used to produce individual "processes" (Krogdahl 2003). In a sense, the activity declaration could be thought of as a platonic idea that describes how a process should be built and function, describing its essential characteristics, and then upon being run, the program could produce multiple instances of the activities that would have concrete values and states associated with the variables described in the declaration (many contemporary OOP languages refer to this same distinction as classes and objects—where a class describes how the object functions and then the program produces multiple objects as instances of the class). Thus, while shifting from an ontological system of computation in which the world consists of two types of object to a single type of object, Dahl and Nygaard bifurcated their ontology along another line into a separation between concept (or activity in their language) and process. These developments were not only a change in metaphors but also produced changes in the underlying code of the compiler. At this point, Nygaard and Dahl decided to abandon the initial plans of creating Simula as a preprocessor for ALGOL, a language that while not particularly well adopted, constituted a major development in the theory of programming languages.[6] As a result of the abstractions they were writing into Simula, the underlying data structure (the way in which information is organized and represented as bits within memory) used by ALGOL was not flexible enough for their purposes. So, they created a completely new compiler that could handle both ALGOL code and data types along with the more flexible data structures that Simula required. Thus, the development of Simula proceeded and was constrained along two fronts. The language required an adequate underlying system for managing the data structure and the central metaphor of processes had to account for systems and objects in the world that were to be described and translated into compilable code. The challenge that confronted Dahl and Nygaard was to create a language that could translate between these two fronts.

In addition to these concerns, one of the goals of Simula I, as a simulation system, was to allow for quasi-parallel processes. Since a simulated system would have multiple processes occurring simultaneously, the language required a method whereby a pseudo-system time would allow the emulation of multiple processes. For example, in the case of simulating an airport, multiple counters would be serving customers simultaneously. Since the machines

that Simula ran on could only process a single instruction at a time, simulating multiple processes required scheduling turns so that the language could approximate simultaneity. While this aspect of Simula was not maintained in many successor languages, it has again become an issue as many large-scale programs are now run on multiple computer cores that can split up tasks and actually compute them simultaneously.[7] Still, this movement towards multiple processes acting independently while influencing each other in the global ecosystem of the program was an important step in moving from a series of commands to a set of objects.

While Simula was not unique in developing in this direction, we can observe in this movement a moment in a larger divergence between math and programming. Despite the initial design plans, they quickly moved away from thinking of the language in terms of mathematics, "We no longer regarded a system as described by a 'general mathematical structure' and instead understood it as a variable collection of interacting processes—each process being present in the program execution, the simulation, as an ALGOL stack" (Nygaard and Dahl 1978, pp. 249–250). This statement is symptomatic of a much larger rupture that has disrupted late twentieth and early twenty-first century science between the primacy of computation and mathematics. Within computation, there are those such as Nygaard and Dahl, who see programming breaking with mathematics, while there are those who stress the mathematical nature of computation.[8] Furthermore, there has been a considerable amount of work in the semantics of programming languages that while recognizing the divergence between programming language and pure mathematics or logic, has attempted to create mathematical descriptions of these computational systems.[9] Likewise, within the physical sciences, some now see a universe that builds chaotic processes out of a near-infinite number of simple processes running and interacting concurrently rather than deep mathematical structures.[10] Math and computation are obviously related, but the question that simultaneously unites the two and inscribes their difference is whether math explains a fundamentally computational world or the other way around. Regardless of which wins out in the end, the important point for our purposes is that in this movement away from being exclusively mathematical, computers quickly came to contain multitudes. While pure mathematical computation will always be possible for computers, increasingly, they have become machines for working on data and computing complex systems out of large collections of relatively simple interactions and processes.

With the transition from a dual ontology to a single ontology of processes and the movement from mathematics to pseudo-parallel interacting processes, Simula I succeeded in the computational ecology of the mid-1960s. Throughout 1965 and 1966, Dahl and Nygaard taught and promoted the language. As its use expanded, they realized that its underlying

architecture could be generalized even further from a simulation language for ALGOL to a general programming language (Holmevik 1994, pg. 32). In developing a second version of Simula, they were heavily influenced by the work of C.A.R Hoare on record classes and subclasses, which he presented at a conference in the summer of 1966 (Dahl 2004).[11] Hoare's work contributed the notion that one could describe a broader data structure and then also describe more specific versions of that data structure. For example, one could describe a number of things about "vehicles" and then also define more specifically programmatic differences between "cars" and "trucks." With this and a number of lessons from Simula I in mind, they commenced developing what would become Simula 67. They began designing the new language and presented a paper at a computer language conference (IFIP TC2) in Oslo in May of 1967. That same month, they signed a contract with Control Data to implement the still-unfinished language. The following month, the Simula 67 Common Base Conference was held to oversee the specification for the language. Under the auspices of the Simula Standardization Group, which was created out of the common base conference, a finalized specification was created and the first compilers were released in 1969 (Holmevik 1994).

While the increased involvement of committees and groups speaks to the growing complexity and interest in Simula, the most important philosophical shift between the first and second versions of Simula was the replacement of processes with objects. Dahl (2004) explains the difference by stating:

> The most important new concept of Simula 67 is surely the idea of data structures with associated operators (and with or without own actions), called objects. There is an important difference, except in trivial cases, between: *the inside view of an object*, understood in terms of local variables, possibly initialising operations establishing an invariant, and implemented procedures operating on the variables maintaining the invariant, and *the outside view*, *as presented by the remotely accessible procedures*, including some generating mechanism, dealing with more "abstract" entities. This difference, as indicated by the car example in Simula I, and the associated comments, underlies much of our program designs from an early time on, although not usually conscious and certainly not explicitly formulated (p. 21). [Italics mine]

For Dahl and Nygaard, two main changes differentiate the process from the object. First, a distinction is drawn between the inside and the outside of the object. In contemporary terms, this is referred to as encapsulation. The inside of the object is protected from other objects

and the global program, and only functions that are part of the object have access to internal elements. For example, in the case of an airport ticketing system, the variable that stores the seat number could only be accessible from within a ticket object. A function could be created to change the seat number that would guarantee that it could only be set to a valid seat and if the change required an upgrade to first class require that the upgrade fee be paid. In complex programs, this helps prevent programming mistakes by allowing objects to guarantee that their internal states are kept within expected limits and to make sure that two parts of a program treat a data structure consistently. Second, with the movement from processes for simulation to a general schema of computational objects, simulation and the quasi-parallel simultaneity of Simula I become special instances of the general scheme.[12] With Simula 67, the simulated time of Simula I was no longer included as a native element of the language but could be created with a "simulation class" that would allow objects to interact in pseudo-parallel.[13]

Thus, the process-based ontology of Simula I was replaced by an object-oriented ontology in Simula 67. With this shift, time drops out as a fundamental ontological component and the object is left to manage its own internal affairs. We can see in this how much the work of creating programming languages is a process of describing the world and crafting a language that can allow the programmer to describe a world of computation. It is a developmental process, wherein certain ideas and metaphors for computation have succeeded and others have fallen aside. Specifically, in the case of the Simula languages, we see a development away from a process-oriented world towards one of objects and explicitly towards objects that are defined by their withdrawal from the world. While processes are entirely exposed, objects hide some of their functioning in order to allow the programmer to separate the work of defining the object's functioning and the interactions with other already encapsulated object. In the few short years between the initial design for Simula and Simula 67, the combined necessities of computation, thought, and language completely rearrange the planned metaphor for describing the world drawing significantly closer to a fully object-oriented mode of programming.

5. The Pre-History of the Object II: Sketchpad

The other early influence on object-oriented programming, Sketchpad, was designed to support computer graphics and became a major precursor to computer-assisted design (CAD). Ivan Sutherland developed Sketchpad a few years prior to the first version of Simula as part of his doctoral thesis at MIT in 1963 under the supervision of Claude Shannon, a pioneer in computation and the founder of modern information theory. Sketchpad was revolutionary on a number of fronts; it essentially founded the field of computer graphics and was a major influence on the graphical user interfaces that came to

dominate desktop computing. Sketchpad allowed a user to draw and manipulate lines and shapes on a computer using a light pen in much the same way that mice and trackpads are used today. Sketchpad was not simply a graphic drawing tool; it was designed to aid in drafting work and attempted to add features to this work by going beyond what was possible by hand. The software allowed a user to create master drawings that could then easily be reproduced and added as instances to another drawing. Hence, if one were designing a circuit, a single master transistor symbol could be drawn and then the actual drawing would link to the master; so if a user updated the master drawing, it would automatically update each instance. Moreover, the software stored the topology of the drawing; such that if a transistor symbol were moved, the software would move the lines connected to it. One can see here the basic outlines of an object-based model in the relationship between master drawings and instances. Thus, while Sketchpad presented a very different programming interface to the user than what is normally thought of as a programming "language," the interaction provided by sketchpad in a sense allows for object-oriented drawing. Each of the subdrawings were encapsulated and left to handle their own affairs, while the "programmer" or drafter is working on another level of the drawing.

According to Sutherland's account of the development of Sketchpad, there was a very similar process to Nygaard and Dahl's in which he began with an attempt to create a certain model in software that was generalized and abstracted over time. In the earliest stages of development, Sutherland worked to implement "strong conditions" that would allow a user to carry out common drafting techniques. Sutherland (2003) says of these early designs, "At this time a notion of 'strong conditions' was used to give geometric nicety the drawing. For example, lines could be drawn parallel or perpendicular to existing lines but carried no permanent trace of the relationship other than the accident of their position. This early effort in effect provided the T-square and triangle capabilities of conventional drafting (pp. 32–33)." While the initial plan was to essentially create digital versions of drafting tools, Sutherland in consultation with his advisor abandoned this plan and instead created a generic structure for defining arbitrary constraints (Shannon also convinced him to include the capability to draw circles in addition to lines). Sutherland (2003) says of this shift,

> Out of these talks came the conviction that a generic structure would be necessary if the system were to be made easy to expand. On June 9, 1962 all this new information came to a head and an entirely new system was begun which has grown with relatively little change into the final version described here. Had I the work to do again, I could start afresh with the sure knowledge that generic structure…would more than recompense the effort involved in achieving them (p. 35).

So, we see here a similar movement towards abstraction; the attempt to emulate a limited set of elements, such as a T-square, is expanded, increasing the conceptual complexity of the language overall but making it more efficient once the central metaphor is understood. Moreover, just like in the case of Simula, Sutherland had to develop a new way to store the raw data for his program in order to link the conceptual innovations of Sketchpad with the underlying structure of the bits. In order to effectively store the topological relations of a drawing, Sutherland invented a ring structure. The ring structure was used to make it easy to update a drawing without having to search for all of the connected points. For instance, one "ring" may consist of all of the lines terminating in a given point; so if the point is moved, the computer simply goes around the ring until it has updated all of the lines. And, each ring is linked to other components by a "chicken" element, "The hen pair is contained within a block which will be referred to, for example, in a point block, while the chicken pair is contained in a block making reference to another, for example, a line block making reference to the point" (Sutherland 2003, p. 41). It is perhaps an unfortunate accident of history, or a result of the decline of American agriculture, that later languages never used this particular terminology of chickens and hens. Regardless, with Sketchpad, as we saw with Simula, the language developed simultaneously along two tracks. The desire for a new metaphor to conceptualize, in this case, drawing required both the creation of a workable generalized metaphor and at the same time the creation of an organizational schema for the raw bits.

Still, this organization of the bits was not created ex nihilo but built on previous abstractions. The most basic of these abstractions are built in at the level of hardware, as computers are designed to operate on "words." A word is the standard number of bits that are treated as an addressable unit by the computer (in contemporary computers, this is most often 32 or 64 bits). So, even machine code operates not directly on bits but on collections of bits. Building up from words, different data structures organize these words into structures that aid in computation. Sutherland developed his ring structure based on *n*-component elements, a method of storing complex data types that consisted of multiple linked simple data types developed by Douglas Ross. Incidentally, Ross's work also happened to be a major influence on Hoare's concept of record classes. As we saw above, record classes were a major influence on the design of objects in Simula 67 (Blackwell and Rodden 2003).

Ross's *n*-component elements were part of a more general scheme that he designed for an abstract data type that he called a *plex*. He initially described a *plex* in 1961 as, "much more powerful than list or tree structures (including them as subcases) and appears to be better suited for the concise representation of the complex interrelations of elements which

constitute a 'problem'" (Ross 1961, p. 147). In comparison with other existing data types, *plexes*, and with them *n*-component elements, were significantly more abstract and could store much more complicated types of data. In the development of this idea, along with many of the other central advances of OOP, there can be seen a major shift away from thinking of computation as either mathematics or simply as a series of processes. Instead, even with Ross' *plexes*, computation began to be presented as an ecology of complex interacting elements.[14]

With the development of Sketchpad and the advent of computer graphics, the creation of separable elements as a means for understanding computation provided a distinct advantage as elements could be manipulated and designed and then ignored in order to work on another level of abstraction. This has served to be an effective way to work with computer graphics, as each element can be treated on its own and the element can control many of its actions internally. For example, in contemporary operating systems, windows, icons, and other elements can be programmed once and then multiple instances can be created. With Simula, creating a set of processes, and then later objects, served as a means to describe a simulation of the world. With Sketchpad, we see the description and creation of a visual world internal to the computer, "At the outset of the research no one had ever drawn engineering drawings directly on a computer display with nearly the facility now possible, and consequently no one knew what it would be like" (Sutherland 2003, p. 28). With the rise of computers as graphical systems with complex interactions, computers now contain worlds onto themselves. The object-like systems that Sutherland created, along with the data structures that allowed for these systems, played a key role in developing the conditions for the construction of these worlds and their presentation to the general population.

6. The Beginning of Object Orientation: Smalltalk

Thus, the digital object ossifies out of two histories, one virtual and another visual. Within computation, the object arises out of a desire to create a model of the world within the computer but at the same time out of an attempt to create a whole new visual world native to the computer. In both attempts at creating a world, the object, or the drawing in the case of Sketchpad, appears as a necessity due largely to increasing complexity and a desire by the designers of these languages to allow elements to handle their own affairs. One of the key organizational benefits of object-oriented programming is the ability to write the functions and data structures for objects and then ignore or forget what is going on inside the object and only function on the purposefully exposed external aspects of the object.

While Simula 67 contained objects and Sketchpad produced object-like subdrawings with topological constraints, Alan Kay saw in objects a whole programming philosophy. It was

this philosophy that he infused into Smalltalk, a language developed by Kay to be completely object-oriented and teach users how to conceptualize programming. The earlier languages offered objects or object-like behaviors as a possible method for controlling programs, but Smalltalk was designed to consist solely of objects that passed messages between themselves. Kay, along with Dan Ingalls and Adele Goldberg who worked on the language and its implementation, raised the notion of objects to an overarching concept that could encapsulate the entire programming system, "I spent a fair amount of time thinking about how objects could be characterized as universal computers without having to have any exceptions in the central metaphor" (Kay 1996, p. 258). Smalltalk built explicitly upon the developments of Simula and Sketchpad but was oriented towards a new age of personal computing.

Millions of potential users meant that the user interface would have to become a learning environment along the lines of Montessori and Bruner; and needs for large scope, reduction in complexity, and end-user literacy would require that data and control structures be done away with in favor of a more biological scheme of protected universal cells interacting only through messages that could mimic any desired behavior (Kay 1996, p. 511).
Much of Kay's work on Smalltalk was aimed at a project to build a small personal computer that children could use to learn how to program. Kay was invested in developing Smalltalk as a language that would be conceptually tight and could be used to teach programming. All of the versions of Smalltalk prior to its 1980 version, known as Smalltalk-80, were not publicly released and kept largely internal to Xerox PARC, where Alan Kay led the Learning Research Group. The first version of Smalltalk was developed in 1971, largely as a proof of concept. Between 1971 and 1980, a number of versions were developed and systems built in the language to make it useable on a machine with bitmap graphic capabilities. Smalltalk-80 was ultimately released publicly, and the entire August 1981 issue of *Byte Magazine* was devoted to the language.

Even more than in Sketchpad and Simula, there is a clear philosophical and pedagogical interest in presenting a unifying metaphor for computation in Kay's writing about Smalltalk. Here, he attests to his philosophical method for conceptualizing computer programming. Philosophically, Smalltalk's objects have much in common with the monads of Leibniz and the notions of 20th century physics and biology. Its way of making objects is quite Platonic in that some of them act as idealisations of concepts—Ideas—from which manifestations can be created. That the Ideas are themselves manifestations (of the Idea-Idea) and that the Idea-Idea is a-kind-of Manifestation-Idea—which is a-kind-of itself, so that the system is completely self-describing—would have been appreciated by Plato as an extremely practical joke (Kay 1996, pp. 512–513).

In this conception of Smalltalk and object-oriented programming, the dual ontologies that were present in both Sketchpad and Simula are flattened to a single ontology, such that even classes are instances of the idea of classes (Smalltalk-80 introduced a second ontological notion, metaclasses, but Kay was less involved at that point in the project and believes they were unnecessary—in these movements, there is not a single evolutionary tendency and the ontological systems proposed are unstable and shifting; Kay 1996). Even platonic ideas became idea-objects. Robson (1981, p. 86) puts this even more succinctly, when describing object-oriented programming in the Smalltalk issue of *Byte*, stating, "In a system that is uniformly object-oriented, a class is an object itself." The class itself becomes merely a specific type of object.

Kay also pushed human language further into the computer by advocating for late binding of variables. Late binding simply means that variables and methods are not directly mapped in the compiled program and instead names are used to reference variables and methods; such that they can be mapped at a later time. The idea of late binding had existed in previous languages such as LISP, but Kay raised it to a principle of OOP. In a discussion of the meaning of OOP he says, "OOP to me means only messaging, local retention, and protection and hiding of state-process, and extreme late-binding of all things" (Kay and Ram 2003). While in many instances late binding slows down the execution of a program, because methods have to be looked up at run-time, it simplifies the process of keeping track of and modifying how a program runs. Moreover, such a design pushes language even further into the operation of the program by maintaining variable names in the compiled code.

Smalltalk also advanced this metaphor of computation as objects along another related track that turned computation into a self-referential system of interacting computers. Kay (1996) describes this understanding of objects in relation to computers:

> Smalltalk is a recursion on the notion of computer itself. Instead of dividing "computer stuff" into things each less strong than the whole—like data structures, procedures, and functions which are the usual paraphernalia of programming languages—each Smalltalk object is a recursion on the entire possibilities of the computer. Thus its semantics are a bit like having thousands and thousands of computer all hooked together by a very fast network. Questions of concrete representation can thus be postponed almost indefinitely because we are mainly concerned that the computers behave appropriately, and are interested in particular strategies only if the results are off or come back too slowly. (p. 513)

Thus, even more than simply reducing computation to a series of objects, computation by way of objects became completely synonymous with objects. Each computer became an object and each object a simulated computer. Kay ultimately attempted to close his own ontological circle and find object-computers recursively defining each other and infinitively deferring any concrete representation. It is striking how much this vision anticipated the current world of networked computers with large cloud-based tasks spread across multiple individual machines.

In order to advance this flat and recursive ontology of computation, objects required a method of interaction. In Kay's metaphor for computation, there could be no transcendental system that could handle the interactions between objects, and so, messaging became important to allow objects to communicate independently. Messages were central to the coherence of the object metaphor of computation, "Smalltalk's design—and existence—is due to the insight that everything we can describe can be represented by the recursive composition of a single kind of behavioral building block that hides its combination of state and process inside itself and can be dealt with only through the exchange of messages" (Kay 1996, p. 512). This insight is central to object orientation; in order to reduce the world to a single building block, which handles its own affairs, there must be a means of communication between the objects.[15] For objects to be possible units of computation, they must have recourse to messages. Messages and objects ossify out of computation-as-procedure simultaneously. There is no object before the relation between them; otherwise, one is left with a null object that refuses the very possibility of computation. While Kay adopted the term "object-oriented programming" in the early 1970s to describe this philosophy of programming, it very well could be called message-oriented programming.[16]

7. Towards a Theory of Objects

Within the object-oriented paradigm of programming, objects were invented for the purpose and benefit of abstracting the real. The computational real is not made of objects or even of "numbers"; rather, the real is comprised of bits. But even the bits are abstractions born of language, underneath them there is merely a difference of voltage, levels of light intensity, or a direction of magnetization. The digital itself is a way to engage analog electrical change over the time of the signal or the space of memory as an object that can be assumed to be stable. From pure electric difference moving through the material stratum of the computer, we forge bits; these bits give rise to numbers and commands. Out of these basic abstractions, more complex abstractions are created through language and metaphor. The computer comes to operate on processes, functions, classes, and objects as a result of its collision with human language. None of these objects, e.g., bits, functions, classes, and even the object that Smalltalk invented, exist as objects outside of language. This is not to say

that they are only linguistic or that they do not really exist. The digital real exists and exerts itself within its ossification into object form, but in the real, there are no objects, merely a vast field of electric difference. Likewise, it is not to say that everything is language or that everything is material. A discourse that does not affect the electronic system within the machine can never flip a bit. Conversely, for electric difference to mean anything, it must be delimited as an object, process, or other data type within language. Programming is a process and system of representing and simultaneously hiding the real in language.

In this movement from the computational real to the object, nothing is hidden a priori. What we could term "objectification," the creation of an object in language out of a series of processes, withdraws and hides part of the object's operation from the programmer. We can have access to everything about the object, yet something must be hidden in order to proceed in the task of abstracting and programming anything but the most trivial or exclusively mathematical system. The status of any given bit can be queried and revealed, but in order to interact with a computer meaningfully and productively requires obfuscating the overwhelming scale and complexity of the individual bits. One of OOP's main advantages is the ability to encapsulate objects and leave their internal affairs to the objects. Apropos Smalltalk, the object is created through a double movement. On the one hand, the object is encapsulated and made to withdraw; simultaneously, interfaces are created such that the objects can exchange messages with other objects. Even in these attempts to flatten the ontology of computation to a single, recursively defined building block, new modes of existence keep arising, classes, messages, metaclasses, etc. At least in terms of computing, objects are neither the "natural" building block of computation nor the inevitable outcome of a process of abstraction or increasing complexity. Rather, they are a negotiated and unstable means of counting the computational real, modeling and creating worlds inside and outside the computer and interfacing computation with human language. Moreover, even if one were to take an evolutionary position on the development of programming languages, it is not clear that the existence of a specific set of objects is the end result of such an evolution. Within an object-oriented programming framework, there is not a "correct" set of objects that one must work with. Furthermore, there is not even a correct way to define what an object is; different ontologies emerge from the logic and metaphors of different languages. The whole task of designing a program involves determining how best to delimit the set of objects and their interfaces that will constitute the program. Thus, both the creators of programming languages and programmers themselves must do ontological work, in so much as a given program must be broken into its ontological components (whether they are objects, classes, procedures, etc.).

Alt (2011, p.292), in suggesting that object-oriented programming was the major advance that turned computers from calculating machines into media, argues, "Encapsulation requires the programmer to conceive the space of the program as embodied, three-dimensional space containing multiple individual subjectivities." It is precisely the process of this creation of these subjectivities that handle their own affairs that is of critical importance in any theory of the object. The creation and encapsulation of these objects place demands on users, programmers, and other subjects from the moment of their creation and delimitation. In sum, a philosophy of objects that would learn from the philosophy and history of OOP must take up as one of its central tasks a serious consideration of the way in which objects are demarcated in language and the implications any set of objects, or ontology, has on the global ecology of objects.

Here, we find ourselves at odds with philosophers who insist on the concrete existence of objects. The most radical interpretation of the history of object-oriented programming leads us to the proposition that even if we accept the reality of bits, or electrical difference, objects, as unique bundles of substance, exist only through language (or something analogous to language—as the object-oriented ontologists' attempt to push philosophy beyond the human is commendable). This is not to say that they are not real, but that their delimitation as unique things exists in language even though the substance of objects is material. Thus, this is not at all an idealist or anti-realist position. There are bits and electric current that are undeniably real, but their organization into objects is a function of language. In this, we arrive at a similar position to the one Smith (1996) presents in his text, *On the Origin of Objects*, in which he draws from Computer Science to claim that objects must be "registered." Likewise, we are very close to Barad's agential realism (2007), inspired by her research in quantum mechanics, in which a scientific or social apparatus makes an agential cut that co-creates both subject and object.

One could argue that perhaps what we see in this history is not the creation of objects as an ontological element but rather the development of a language to such a level of complexity that it is finally able to describe objects that already exist in the world outside of language. Against this objection, it must be noted that what we witness in this history, what drives it forward, is the difficulty of relating language to world, whether it is a simulated world or a new world of computer graphics. Moreover, the examples presented here and in the textbooks and manuals for object-oriented programming demonstrate the often-difficult art of defining the objects that will constitute a given system. Not only are individual objects unstable but even the very concept of what an "object" is, how they are to function, and their relation to the rest of the system are unstable and negotiated through the construction of these various languages.[17] With object-oriented programming, we are confronted with a

whole series of emergent ontologies rather than a single notion of objects that is merely to be filled in.

The difficulties of developing and using these languages to describe a problem speak to the non-predetermined boundaries of the computational real. In fact, one of the major programming problems that can compromise computer systems' security is known as a buffer overflow. Buffer overflows happen when the program expects a variable to be smaller than a certain size, but a larger number or string is inserted into the variable. The program keeps writing the entire data, overflowing the intended variable and depositing bits into other variables, potentially causing programs to crash. Because programs store data and commands in the same place, buffer overflows can be used maliciously to take over a machine by inserting new commands into the memory locations that will be executed. Thus, even the primitive data types of computation are subject to overflowing their bounds since the boundaries are a convention of language and not inscribed directly into the computational real. Even these basic objects of computation leak and overflow. It is only through mechanisms within programming languages themselves that these boundaries can be defined and secured. Language must secure its own boundaries.

8. Conclusion: Object Orientations and Their Discontents

We arrive then at a notion of object orientation that is markedly different from that advocated by object-oriented ontologists. While there are any number of flavors of new realisms and materialisms, the history of OOP seems to have the most to say about those who identify their philosophy with a realism founded on the solidity of objects, like Harman. For Harman, the founder of object-oriented ontology, the world is made up of objects and the task of an effective philosophy is to speculate about the implications of their interactions with and without human intervention. In an argument based upon his reading of Heidegger, he claims that objects have depth and are withdrawn from each other preventing anyone or anything from accessing their totality. Harman (2011, p. 22) is explicit about his object orientation and the existence of objects stating, "the 'object-oriented' part of the phrase is enough to distinguish it from the other variants of speculative realism. By 'objects' I mean unified entities with specific qualities that are autonomous from us and from each other." Likewise, while Meillassoux (2008) does not identify himself as an object-oriented philosopher, his entire argument in *After Finitude* consists of a thought experiment founded on the reality of a bounded and finite "ancestral" object. While there are a number of debates internal to speculative realism and its offshoots, which are too numerous and detailed to recount here, this fidelity to the notion of objects in Harman and other's thought is squarely at odds with the underlying philosophy and task of object-oriented programming.

From those dedicated to object-oriented philosophy, little has been said about object-oriented programming and the philosophical claims of its creators. One of the only connections between these two philosophies is an article critical of both written by Galloway (2013). Galloway claims that object-oriented philosophy (and with it a number of philosophies built on mathematical foundations, such as Badiou and Mellasioux's work) is politically implicated in its structural complicity with object-oriented programming, which he sees as the central technology underwriting global capitalism. Galloway correctly points out how apolitical many of the object-oriented philosophers tend to be in their attempts to separate ontology from politics. Moreover, even when they make political claims, they tend to amount to little more than saying that we will be able to somehow develop a politics addressing our debt to our surroundings if we treat objects seriously. Beyond expanding the social to include objects, they offer little in terms of how to operate in this society of objects or how one would develop a political or ethical system out of it. Despite Galloway's recognition of this anti-political tendency, it seems clear in the light of the history of object-oriented programming that there is in fact a large gulf between object-oriented programming and object-oriented philosophy.

To take OOP seriously as a philosophical and ontological endeavor requires a very different form of philosophy than the present object-oriented ontology. If we understand objects to be an abstraction of the real that ease our ability to understand and live within its complexity, all the while obfuscating the underlying processes, it becomes possible to envision a form of object-oriented philosophy that would seek not to understand what objects exist but how and to what ends the real could be described as objects, processes, or other ontological categories. Instead of accepting as given the boundaries between objects, an object-oriented philosophy that follows the lessons of OOP must explore both our popular understanding of objects and new possible segmentations of the real. While such a project could take a number of turns, perhaps the Marxist critique of objectification and the solidification of social relations into commodities could provide an inspired starting point for a philosophy that would explicitly call out the need to be aware of the political nature and social implications of any ontological system and its assumptions about the solidity of objects and attributes. Against the solidity of objects gaining popularity in certain quarters, a philosophy informed by the genealogy of the digital object must confront and understand the multitude of ways in which the real can be divided, described, and understood.

References

Abadi, M. and Cardelli L. (1995). An imperative object calculus. TAPSOFT'95: Theory and Practice of Software Development, 469–485.

Abadi, M., & Cardelli, L. (1996). *A theory of objects*. New York: Springer.
Alt, C. (2011). Objects of our affection: how object orientation made computers a medium. In E. Huhtamo & J. Parrika (Eds.),*Media archaeology: approaches, applications, and implications*. Oakland: University of California Press.

Barad, K. (2007). *Meeting the universe halfway: quantum physics and the entanglement of matter and meaning*. Durham: Duke University Press.

Blackwell, A. and Rodden, K. (2003). Preface in Sutherland, (pp. 3–6).

Bloch. (2001). *Effective Java*. Boston: Addison-Wesley.

Dahl, O. (2004). The birth of object orientation: the Simula Languages in from object-orientation to formal methods, lecture notes in computer science, 2635 of the series Lecture Notes in Computer Science, 15–25

Dean, J., & Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. *Communications of the ACM, 51*(1), 107–113.

Derrida, J. (1993). Circumfessions in Bennington, G. and Derrida, J. Jacques Derrida. Chicago: University of Chicago Press.

Evens, A. (2006). Object-oriented ontology, or programming's creative fold. *Angelaki: Journal of Theoretical Humanities, 11*(1), 89–97.

Galloway, A. (2004). *Protocol: how control exists after decentralization*. Cambridge: MIT Press.

Galloway, A. (2006). Language wants to be overlooked: on software and ideology. *Journal of Visual Culture, 5*(3), 315–331.

Galloway, A. (2013). The poverty of philosophy: realism and post-Fordism. *Critical Inquiry, 39*(2), 347–366.

Goetz, B. (2004). Java theory and practice: fixing the Java memory model, Part 2. IBM.https://www.ibm.com/developerworks/library/j-jtp03304/j-jtp03304-pdf.pdf

Golumbia, D. (2009). *The cultural logic of computation*. Boston: Harvard University Press.

Harman, G. (2011). On the undermining of objects: grant, Bruno, and radical philosophy. In

L. Bryant, N. Srnicek, and G. Harman (Eds.), The speculative turn: continental materialism and realism (pp. 21–40). Victoria: re. Press, 2011.

Hayles, K. (1999). *How we became posthuman: virtual bodies in cybernetics, literature, and informatics*. Chicago: University of Chicago Press.

Holmevik, J. (1994). Compiling SIMULA: a historical study of technological genesis. *IEEE Annals of the History of Computing, 16*(4), 25–37.

Kafura, D., & Lavender, G. (1993). Concurrent object-oriented languages and the inheritance anomaly. In M. Quinn (Ed.), *Parallel computers: theory and practice* (pp. 221–264). New York: McGraw-Hill.

Kay, A. (1996). *The early history of Smalltalk in history of programming languages II (pp. 511–598)*. New York: Association for Computing Machinery.

Kay, A. and Ram, S. (2003). On the meaning of "object-oriented programming." E-mail exchange. http://userpage.fu-berlin.de/~ram/pub/pub_jf47ht81Ht/doc_kay_oop_en . Accessed: 15 December 2015

Kittler, F. (1995). There is no software. *CTheory, 10*(18), 147–155.

Kittler, F. (1999). *Gramophone, film, typewriter. Translated by Winthrop-Young, G. and Wutz, M*. Stanford: Stanford University Press.

Krogdahl, S. (2003). The birth of Simula. HiNC 1 (History of Nordic Computing) Conference.

Manovich. (2013). *Software takes command*. New York: Bloomsbury.

Milner, R. (1999). *Communicating and mobile systems: the π-calculus*. Cambridge: Cambridge University Press.

Nygaard, K., & Dahl, O. (1978). The development of the SIMULA languages I. *ACM SIGPLAN Notices, 13*(8), 245–272.

Meillassoux, Q. (2008). *After finitude: an essay on the necessity of contingency. Translated by Brassier, R*. London: Continuum.

Priestley, M. (2011). *A science of operations*. New York: Springer.

Robson, D. (1981). Object-oriented software systems. *Byte Magazine, 6*(8), 74–82.

Ross, D. (1961). A generalized technique for symbol manipulation and numerical calculation. *Communications of the ACM, 4*(3), 147–150.

Salomon, D. (1993). *Assemblers and loaders*. West Sussex: Ellis Horwood. Available:http://www.davidsalomon.name/assem.advertis/asl.pdf .

Smith, B. (1996). *On the origin of objects*. Cambridge: MIT Press.

Sutherland, I. (2003). *Sketchpad: a man–machine graphical communication system*. Cambridge: University of Cambridge.

Thacker, E. (2004). *Foreword: protocol is as protocol does in Galloway, a. protocol: how control exists after decentralization (pp. xi-xxii)*. Cambridge: MIT Press.

White, G. (2004). *The philosophy of computer languages in Floridi, L. The Blackwell Guide to the Philosophy of Computing and Information (237–247)*. Oxford: Blackwell Publishing.

Wolfram, S. (2002). *A new kind of science*. Champaign: Wolfram Media.

Yares, E. (2013). 50 years of CAD. Design World, February 13, 2013

Endnotes

[1] Abadi and Cardelli (1996) outline a taxonomy that attempts to account for the variety of object-oriented languages while suggesting a unifying system for explaining them. The complexity of this taxonomy and the changes that "objects" have undergone over the course of their history suggest that the very concept of an object is not fixed and must be worked out in language.

[2] Krogdahl (2003) provides an extensive history of Simula. Priestley (2011) traces the history of both Simula and Smalltalk. Manovich (2013) explores the history of Kay's work on Smalltalk with a focus on the development of

the Dynabook. Kay (1996) provides one of the most comprehensive overviews of the history of Smalltalk. It appears that little has been written about the history of Sketchpad. While informative, what has been written has largely traced the history of computer graphics, such as Yares (2013). Additional histories, many of the most instructive for present purposes written by those involved in designing these languages, are cited below.

[3] Smalltalk is especially fruitful as an object of study in this regard, as Kay and his collaborators strove to maintain a philosophical coherence to their language that reflected a complete theory of how computing could be understood. Priestley (2011) notes that many later object-oriented programming languages, such as C++, abandoned the rigorousness of Kay's work in order to include elements of earlier styles of programming.

[4] In addition to the below discussion of the negotiations and changes surrounding the development of Simula, Sketchpad, and Smalltalk, see for example the changes made to the Java Memory Model in Goetz (2004).

[5] I use the term "real" to describe the material aspects of computation following Kittler's Lacanian-influenced media-historic materialism (Kittler 1999).

[6] Priestley (2011, 225) suggests that while Algol was not very successful in practical terms, "what changed the face of programming was not simply the Algol 60 language but rather a coherent and comprehensive research program within which the Algol 60 report had the status of a paradigmatic achievement, in the sense defined by the historian of science Thomas Kuhn."

[7] See, for example, Dean and Ghemawat (2008).

[8] Priestley (2011) argues that Algol was exemplary in attempting to unify a logical and mathematical structure of programming that Smalltalk ultimately broke with. Moreover, there are still major developments in computer science and programming that stress the mathematical nature of programming, for instance, Milner's (1999) π-calculus or Abadi and Cardelli's description of an "object-calculus" (1995). Likewise, the growing importance of encryption for computing has relied heavily upon and even pushed the development of number theory. While all of these suggest an intimate relation between programming and mathematics, the development of these languages and the purposeful abandonment of the language of mathematics in Simula and also later in Smalltalk (Priestley, 2011) point to at the very least a conception of programming that moves away from mathematics, even if it could potentially be recuperated.

[9] The mathematical study of the semantics of programming languages functions as a possible "third way" between mathematical and computation practices by attempting to mathematically define the emergent structures of computation based on various programming languages. Moreover, the semantic descriptions of programming languages have then been used to attempt to design future languages. In addition to Milner and Abadi and Cardelli's work (fn. 8), see White (2004).

[10] Stephen Wolfram is one of the most vocal proponents of this notion of a computational rather than a mathematical universe. For instance, see Wolfram (2002).

[11] Hoare's presentation at the NATO Vilard-de-Lans Summer School, where Dahl and Nygaard also presented their work on Simula I is available: C.A.R. Hoare, "Record Handling," http://archive.computerhistory.org/resources/text/knuth_don_x4100/PDF_index/k-9-pdf/k-9-u2293-Record-Handling-Hoare.pdf

[12] While many of the ideas developed in Simula were aimed at concurrent processing, they were likely abandoned in part because of the difficulty of constructing systems where objects can potentially end up dead-locked, as in the dining philosopher's problem, waiting for other objects who are waiting on them. Despite the movement away from concurrency and simulation in many object-oriented programming languages, there is a continued interest in the development of concurrent object-oriented programming languages. For a description of some approaches and issues related to concurrency in OOP, see Kafura and Lavender (1993).

[13] Dahl (2004) spells out an example of how this could be implemented.

[14] Ross (1961) even suggests that one can use the control point in a program's execution as a way of storing information as there are certain parts of the program that can only be reached if certain conditions are true. Thus, in a way, the temporal execution of the program becomes spatialized into a geography of information.

[15] As Priestley (2011) argues, this decision to rely on messages further moved computation away from mathematics as the interpretation of a given message was left up to the object that received it. For example, a number could interpret "+" as mathematical addition, while a string could interpret it as concatenation (1 + 2 would return 3, while "a" + "b" would return "ab").

[16] Kay discusses this decision in Kay and Ram (2003).

[17] For example, a major element of many object-oriented programming languages is the notion of inheritance, which allows a programmer to specify modified versions of a class. For example, one could define a class for "animal" of which a "cat" would be a subclass, but it would inherit certain properties from the vehicle. Hoare suggested such a model as part of his record classes (see fn. 10), and Simula 67 contained "prefix classes," which functioned roughly in this manner. While inheritance can greatly ease the amount of work that goes into creating programs, it risks breaking encapsulation (see Bloch 2001, item 14, for a technical explanation of this relationship), in so much as a class, and the objects created from it now rely on code that exists elsewhere. In short, external taxonomies in language (in so much as inheritance creates a taxonomy of classes and subclasses) ultimately reach inside and affect the internal functioning of objects. Thus, through the very demands of writing and language, these objects are pulled outside of themselves, and we see the threat to autonomy and solidity that both define and besiege objects.