1
The Obscure Objects of Object Orientation

Matthew Fuller and Andrew Goffey

Object orientation names an approach to computing that views programs in terms of the interactions between programmatically defined objects – computational objects – rather than as an organized sequence of tasks embodied in a strictly defined ordering of routines and sub-routines. Objects, in object orientation, are groupings of data and the methods that can be executed on that data, or *stateful abstractions*. In the calculus of object-oriented programming, anything can be a computational object, and anything to be computed must so be, or must be a property of such. Object-oriented programming is typically distinguished from earlier procedural and functional programming (embodied in languages such as C and Lisp respectively), declarative programming (Prolog) and more recently component-based programming. Some of today's most widely used programming languages – Java, Ruby, C# – have a decidedly object-oriented flavour or are explicitly designed as such, and whilst as a practice of programming it has some detractors it is deeply sedimented in both the thinking of many computer scientists and software engineers and in the multiple, digital-material strata of contemporary social relations.

This essay explores some aspects of the turn towards objects in the world of computer programming (a generic term that incorporates elements of both computer science and software engineering). It asks what powers computational objects have, what effects they produce and, more importantly perhaps, how they produce them. Seeking to situate the technical world of computer programming in the broader context of the changing composition of power within contemporary societies, it suggests a view of programming as a recursive figuring out of and with digital materials, compressing and abstracting relations operative at different scales of reality, composing new forms of agency and the scales on which they operate and create through the formalism of algorithmic operations. This essay thus seeks to make perceptible what might be called the territorializing powers of computational objects, a set of powers that work to model and re-model relations, processes and practices in a paradoxically abstract material space.[1]

Computation has seen broad and varied service as a metaphor for cognitive processes (witness the ongoing search for artificial intelligence), on the one hand, and as a synecdoche of a mechanized, dehumanized and alienated industrial society on the other – flip sides of the same epistemic coin. As such it might appear somewhat divorced from the rich material textures of culture and a concern with the ontological dimension of 'things'. Indeed, with its conceptual background in the formalist revolution in mathematics initiated by David Hilbert, computing may not seem destined to tell us a great deal about the nature of things or objects at all. In the precise manner of its general pretension (insofar as one can talk about formalism 'in general') to universality, to be valid for all objects (for any entity whatever, in actual fact), formalism is by definition without any real object, offering instead a symbolic anticipation of that which, in order to be, must be of the order of a variable. Objects and things, the variety of their material textures, their facticity, tend to transmogrify here into the formal calculus of signs.

Little consideration has been given to the details of the transformative operations that computer programming – always something a bit more sophisticated than simple 'symbol manipulation' – is supposed to accomplish, or to the agency of computational objects themselves in these transformations. In this article, we take up this question through a brief consideration of object-oriented programming and its transformative effects. We read computational formalism through the techniques and technologies of computing science and software engineering, to address object orientation as a sociotechnical practice. As such a practice, one that bears a more than passing resemblance to the kinds of means of disciplining experimental objects and processes that are described by Andrew Pickering, the effective resolution of a problem of computation is a matter of the successful creation, through programming, of a more or less stable set of material processes – within, but also without, the skin of the machine.[2]

Languages of Objects and Events

To understand the transformative capacities of computational objects entails, in the first instance, a consideration of the development of programming languages, for it is with the invention of programming languages that the broad parameters of how a machine can talk to the outside world – to itself, to other machines, to humans, to the environment and so on – are initially established. Languages for programming computers, intermediating grammars for writing sets of statements (the algorithms and data structures) that get translated or compiled into machine-coded instructions that can then be executed on a computer, are unlike what are by contrast designated as natural languages. They are different not only in the sense that they have different grammars, but also in being designed, in a specific context, as a focused part of a particular set of sociotechnical arrangements, a constellation of forces – machines, techniques, institutional and economic arrangements and so on. A programming language is a carefully and precisely constructed set of protocols established in view of historically, technically, organizationally *etc.* specific problems. Usually designed with a variety of explicit considerations – mostly technical but sometimes aesthetic – in mind, programming languages themselves nevertheless register the specific configuration of assemblages out of which they emerge and the claims and pressures that these generate, even as they make possible the creation of new assemblages themselves. The computer scientist, it might be said, 'invents assemblages starting from assemblages which have invented him [*sic*] in turn'.[3]

The project of object orientation in programming first arises with the development of the SIMULA language. SIMULA was developed by Kristen Nygaard and Ole-Johan Dahl at the Norwegian Computing Centre in the early 1960s.[4] As its name suggests, it aimed at providing a means both to describe – that is, program – a flow of work, and to simulate it. The aim of such simulation was to bring the capacity to design work systems – despite their relative technical complexity – into the purview of those who made up a workplace. As such, the project had much in common with other contemporaneous developments in higher-level computing languages and database management systems, which aimed to bring technical processes closer to non-specialist understanding.[5] Equally, they were a way of bringing formal description of the world out from under the skin of the computer. SIMULA aimed to bring the expert knowledge of the programmer into alliance with the decision-making systems of the workplace in which a social democratic version of workers' councils[6] guided the construction of the staging of work. This tendency arose from what would later become known as participatory design, but it also had its roots in a version of operational research,[7] in which the analysis of work was carried out in order to reduce stressful labour.[8]

The first version of SIMULA, SIMULA 1, was developed not with a view to establishing object orientation as a new format for programming languages per se but rather as an efficacious way of modelling the operation of complex systems. Simulation of work processes becomes desirable as systems develop to a level of complexity that makes understanding them a non-trivial task, and in SIMULA, it was a task that was initially understood as a series of ‘discrete event networks’[9] in which inventory, queuing, work and materials processes could be modelled, and in which there would be a clear correlation between the features of the work process and the way in which it was modelled as an ensemble of computational objects. To produce such epistemically adequate models of complex processes, the tacit ontology of discrete event networks utilized by SIMULA (the ‘network’ was eventually dropped) was one in which real-world processes were understood in terms of events – or actions – taking place between entities, rather than as permanent sets of relationships between them. The language was supposed itself to force the researcher using it to pay attention to all aspects of the processes being considered, directing such attention in appropriate ways.

Initially the language had little impact for general programming; indeed, it wasn't until SIMULA67 was developed that the technical feature that has since been so important to object-oriented programming – the capacity to write programs in which entities combine data and procedures and retain ‘state’[10] – was first sketched out, in terms of the possibility of specifying ‘classes’ and ‘subclasses’, each of which would be able to initiate or carry out particular kinds of actions. SIMULA itself did not really see full service as a programming language, because the computing resources required to enable it to do the work required of it meant that it was not especially effective. At such an early stage in the history of programming and programming languages, the possibility of having a language that would be materially adapted to describing real-world processes inevitably rubbed up against the practical obstacle of writing efficient programs.

However, the innovations of SIMULA in the matter of providing for structured blocks of code – called classes – that would eventually be instantiated as objects – were taken up a decade and a half later in the development of C++, a language developed in part to deal with running UNIX-based computational processes across networks,[11] something which later also became a driver in the development of a further object-oriented programming language, Java. Its tacit ontology of the world as sequences of events (or actions), the design principles built into it, its links with a different imaginary and enaction of the organization of labour, are indicative of a possibility of a programming that is yet to come.

A variant history of object orientation can be told through the development of the Smalltalk language at Xerox PARC, Palo Alto, in the 1970s and 1980s, under the leadership of Alan Kay.[12] Smalltalk frames the nature of objects through the primary question of messaging between the entities or objects in a system. In fact, it is the relations *between* things that Kay ultimately sees as being fundamental to the ontology of Smalltalk.[13] Objects are generated as instances of ideal types or classes, but their actual behaviour is something that arises from the messages passed from other objects, and it is the messages (or events; the distinction is relatively unimportant from the computational point of view) that are actually of most importance. Here, although the effective ordering in which computational events are executed is essentially linear, such an ordering is not rigidly prescribed and there is an overall sense of a polyphony of events and entities in dynamic relation. Kay adduces a rationale for the dynamic relations that Smalltalk sought to construct between computational objects by referring to the extreme rigidity and inflexibility of the user interfaces that were available on the mainframe computers of the time: an approach organized around computational objects allows for a more flexible relationship between the user and the machine, a relationship that would later be glossed by one of Kay's colleagues as allowing for the creative spirit of the individual to be tapped into.[14] This flexible relationship to dynamically connecting objects is part and parcel of a utopian vision of computing – eventually materializing in the PC – in which control is portrayed as being wrested from the institutions in which computers (typically mainframe machines) exist in favour of a new form of ‘personal mastery’.

Critically, for Smalltalk, it is the interactive quality of computation that comes to the fore; a major advance on the rigidly prescriptive order of task execution in extant languages. More specifically, the computational object in Smalltalk is viewed in terms of its capacity to be integrated into a system of learning – one that, following Piaget, Vygotsky and others, is essentially constructivist.[15] As such, the operational brief of the object is not just to function well within the domain of the program, but also to interface with, prompt and sustain – but not, perhaps, to shape (at least, not knowingly) – the learning processes of the user. As with the emphasis on messaging between many lightly but precisely defined objects as the basic structure of the program, Smalltalk emphasizes such interaction, the relations between objects, and between objects' messages and users.

The historical point to be made here is that the link between Smalltalk and interaction-based learning marks something of a shift from the attempts at simulating workplace knowledge evident in the early version of SIMULA, but it is a shift in which there are continuities and discontinuities. Whilst the purposes to which programming is directed are different, the ambivalence that SIMULA hinted at – a program as an epistemically adequate model of some process or set of processes, and a program as a materially effective set of processes in its own right, which might be glossed as the *analytic* and *synthetic* functions of computation – remained in place with Smalltalk. A brief explanation is thus in order.

Discussing the development of Smalltalk, Kay says: ‘object-oriented design is a successful attempt to qualitatively improve the efficiency of modelling the ever more complex dynamic systems and user relationships made possible by the silicon explosion'.[16] Modelling here cannot really be understood as a simple representation, because when the complex dynamic systems and user relationships that object-oriented design models are those that are *made possible* by the silicon explosion, we are moving into a zone of ontological artifice: not simply the acquisition of a knowledge of the world through artefacts (as in economist and artificial intelligence theorist Herbert Simon's conception of the sciences of the artificial and as exemplified in the initial project of SIMULA), but rather a matter of creating models of things that don't otherwise exist. The distinction is important although often ignored, because it points towards the limitations of exploring computational objects in the epistemological terms of representation. Understanding computation from the latter point of view is rather widespread; indeed, Kay himself suggests: 'everything we can describe can be represented by the recursive composition of a single kind of behavioural building block that hides its combination of state and process inside itself and can be dealt with only through the exchange of messages'.[17] On this count, a computer program written in an object-oriented programming language develops a kind of intensional logic, in the sense that each of the objects that the program comprises has an internal conceptual structure that determines its relationship to what it 'refers' to, suggesting that objects are, in fact concepts, concepts that represent objects in a machinic materialization of a logical calculus.[18] As concepts materialized in the physical space/time of effective computational processes, but referring to objects outside machine space, Smalltalk objects clearly indicate the 'analytic' function of computation, but insofar as the Smalltalk software is designed as something to be interacted with in its own right, its 'synthetic' function is equally evident.

So, on the one hand, we have the manner in which object-oriented programming purports to enable the decomposition of the processes that it seeks to model into computationally adequate 'representations' of entities (=objects). On the other, the simple fact that a computational object is an entity in its own right; an entity with a specific, material form, which comes into composition with other entities – whether computational or not – that may or may not find some 'representation' materially in that software. It is, we would suggest, this second aspect of computational objects – modelling behaviour *with* objects, as opposed to modelling behaviour *of* objects – that needs to be understood more precisely.[19]

Abstraction, Errors and the Capture of Agency

It is often argued that with the development of object-oriented programming, a new era of *interaction* between humans and machines was made possible. There is some truth in this, not least because of the way that the architecture of relations between objects obviates the need to have a program structure in which the order of actions is rigidly prescribed. However, the notion of interaction, as descriptive of a *reciprocal* relation between two independent entities, is sometimes insufficient when trying to understand the historical genesis and the peculiar entanglement of relations between computers and humans. It would be more appropriate to consider these relations in terms of a series of forms of the abstractive creation and *capture* and codification of agency: a click of the mouse, a tap of the key, data input, affective investments *etc*. By referring to the creation and capture of agency we seek to underline two things: firstly, that computational objects do not simply or straightforwardly tap into pre-formed capacities or abilities, but instead generate new kinds of agency, kinds of agency which may be similar to what went before but which are nevertheless different (a typewriter, a keyboard and a keypad, for example, capture the agency of fingers in subtly different ways); secondly, that the agency that is created is so as part of an asymmetric relation between human and computer, a kind of cultivation or inculcation of a *machinic habitus*, a set of dispositions that is inseparable from the technologies that codify it and give it expression. It would take too long a diversion to examine in detail how and why this asymmetry exists, but it is crucial to developing an appropriately concrete understanding of the sociotechnical quality of contemporary relations of power.

Part of the rhetoric of interactive computing insists on the ‘intelligent’, ‘responsive’ nature of computational devices, but this obscures the dynamics of software development and the partial, additive quality of the development of interactive possibilities. Computers too, are not very good at *repairing* interactions. They tend, in use, to be considerably more intransigent than their users, bluntly refusing to save a file or open a webpage, or even closing down altogether. Whilst system crashes may not be as common now as they once were, the everyday experience of the development of human–computer interactions has been one in which humans have been obliged to spend considerable amounts of time learning to think more like computers, developing workarounds, negotiating with and adapting to computational prescription. 'Bugs' have, in this sense, played an important role in setting up the asymmetric relations between humans and machines. Relatively speaking, humans will adapt to – or at least learn not to notice – the stupidities of the computer much more quickly than the computer will to humans, in part simply because the time between software releases (with bug fixes) is much greater than that between individual interactions with an application. This asymmetry suggests that there is something of a strategic value to the stupidity of machines, a stupidity that gives machines a crucial role in *modelling* the user with which they interact.[20]

In any case, and *pace* Kay, a computational object is a *partial* object: the properties and methods that define it are a necessarily selective and creative abstraction of particular capacities from the thing it models, and which specify the ways that it can be interacted with. The objects – text boxes, lists, hyperlinks and so on – that populate a webpage, for example, define more or less exactly what a user can do with them. This is a function not just of how the site has been designed and built but, more importantly, of a range of previously defined sets of coded functionality. There is a history to each of these objects and their development,[21] which means that the parameters for interaction are determined by a series of more or less successful *abstractions* of a peculiarly composite, multi-layered and stratified kind.

It is important to note here, though, that abstraction is a contingent, *real* process. The taken-for-granted ways in which humans now interact with machines is the product of material arrangements that do not always obtain.[22] More pointedly, these real abstractions are concretely and endlessly re-actualized by the interactions between and with the computational. Such processes of abstraction might be better understood as forms of *deterritorialization* – in Deleuze and Guattari's sense – a process which always involves ‘at least two terms’ in which ‘each of the two terms reterritorializes on the other. Reterritorialization must not be confused with a return to a primitive or older territoriality: it necessarily implies a set of artifices by which one element, itself deterritorialized, serves as a new territoriality for another.’[23]

Considered in these terms, the capture of agency links the formal structuring of computational objects to the broader processes of which they are a part, allowing us in turn (1) to specify more precisely that the formal structuring and composition of objects has a vector-like quality; and (2) to attend more directly to the correlative feature of reterritorialization. Abstracting *from*, in this sense of a real process, is equally an abstracting *to*: an abstraction is only effective on condition that it forms part of another, broader set of relations, in which and by which it can be stabilized and fortified.

We now turn to a more direct consideration of these issues.

Stabilizing the Environment

In theory, anything that can be computed in one programming language can also be computed in another: this is one of the lessons of Turing's conception of the universal machine. What that means more prosaically for the case in hand is that an object-oriented programming language provides a set of *design* constraints on the engineers working with it, favouring specific kinds of programmatic constructs, particular ways of addressing technical problems, over others. The existence of such design constraints is particularly important when trying to consider the dynamics governing the material texture of software culture. The question then becomes this: given the way in which the asymmetries in human–machine relations enable the capture of agency, is there a way in which the propagation and extension of those relations can be accounted for? Can something in the practices of working with computational objects be uncovered that might help understand this dynamic?

One feature that is associated in particular with object-oriented programming is the way that it is argued to facilitate the *re-use* of code. Rather than writing the same or similar sets of code over and over again for different programs, it saves time and effort to be able to write the code once and re-use it in different programs. Re-usability is not unique to object-oriented programming: it is implied in the features of any kind of programming language, in the mundane sense, for example, that whenever one initializes a variable of a given sort within a routine (say a floating point number), the compiling of the code will allocate a value to an address in memory. That is not something the programmer has to do himself or herself; the routinization and automation of computational tasks imply it as a basic feature of operation. But object-oriented programming favours the re-usability of code for computationally abstract kinds of entity and operation – in other words, for entities and operations that are more directly referent to the interfacing of the computer with the world outside. *Class libraries* are typical of this. Although not unique to object-oriented programming languages, they provide sets of objects, with predefined sets of methods, properties and so on, that find broad use in programming situations: in the Java programming language, for example, the Java.io library contains a '*File*' object, which a programmer would utilize when a program needs to carry out standard operations on a file external to the program (reading data from it, writing to it and so on). Code re-use in general suggests that the contexts in which it is situated, the purposes to which it is put, the interactions to which it gives rise and the behaviours it calls forth are relatively regularized and stable. In other words, it suggests that typical forms of software have found and taken part in the making of their ecological *niches*.

The possibility of re-usability here must thus be understood from two angles simultaneously: (1) as something given specific affordance within the structure of an object-oriented language; and (2) as something that finds in its context the opportunity to take root, to gain stability, to acquire a territory. It was suggested in the previous section that the simple dynamics of adaptation or habituation might account for the latter. The former can be located in the technical features of object-oriented programming languages. We will address this issue first before moving on to a broader consideration of stabilizing practices.

One of the main features of object-oriented programming, distinguishing it from others, is the use of *inheritance*. A computational object in the object-oriented sense is an instantiation of a *class*, a programmatically defined construct endowed with specific properties and methods enabling it to accomplish specific tasks. These properties and methods are creative abstractions. Inheritance is a feature that is often – albeit erroneously – characterized semantically as an 'is a' relationship: a Persian or a Siamese *is a* cat, a savings account *is an* account *etc*. The relation of inheritance defines a hierarchy of objects, often referred to in terms of classes and subclasses. How that hierarchy should be understood is itself a complex question (despite attempts to formalize the relation of inheritance by computer scientists). Critically, though, the relation of inheritance allows programmers to build on existing computational objects with relatively well-known behaviour by extending that behaviour with the addition of new methods and properties.

The relation of inheritance implies a situation in which objects extend and expand their purview, their territory, through small variations, incremental additions, that confirm rather than disrupt expectations about how objects should behave. To put it crudely, it is easier to inherit and extend, to *assume* that small differences are deviations from a norm, than it is to consider that such variations might be indices of a different situation, a different world. Modelling behaviour through the technical constraint of inheritance stabilizes the relations and practices captured in software.

Design patterns extend this logic of code re-usability to the situation of a more complex set of algorithms designed to address a broader problem. The notion of the design pattern is borrowed from architect Christopher Alexander, for whom such patterns are the description of problems that 'occur over and over again in our environment'.[24] A design pattern in software provides a reusable solution to the problem posed by a computational context (we will give an example of this shortly), and whilst such a pattern is obviously a technical entity, it is also a partial translation of a problem that will not originally be computational in nature. This is what makes it interesting, because its very existence is evidence of the increasing complexity that computational abstractions are required to address.

A business information system that is designed to keep track of stock, for example, based on a ‘just-in-time’ model of stock control will create design problems entailing a set of relations among computational objects that are rather different to a system based on more traditional models of stock control (such as amassing large amounts of uniform items at lower unit cost), because the system needs to do rather different things. It implies a variant set of relations between software and users, perhaps entailing an automated set of links between one company and companies further up the supply chain.

In one respect, design patterns respond to the core difficulty of object-oriented software development: the analysis, decomposition and modelling of what a program has to do into a set of objects with well-defined properties. Indeed, that is traditionally how design patterns are understood by software engineers. But their very existence itself is interesting because they provide evidence not just of the growing complexity of the computational environment, but also of its stability and regularity – qualities that such patterns in turn produce. Such material presuppositions are not normally considered in discussions of object-oriented programming (or indeed any programming at all), where the self-evident value and common-sense obviousness of thinking in object-oriented terms are generally shored up in textbooks by analogy to the obviousness of objects themselves; for instance, listing all the rather stable objects – tables, chairs, papers, books and the shelves they sit on – in the author's office.[25] And yet the stability of an environment is absolutely critical to enabling computational objects to exert their powers effectively.[26] However, the pedagogical emphasis on the relatively simple – and the consequent attention accorded to the primacy of the epistemological dimensions of computational objects – does not, as we have argued, really do justice to the processes at work in the historical development of software culture. In particular, it does little to help us understand the ways in which agency is not only captured but also configured and stabilized in predictable, routine patterns. It would be more appropriate, perhaps, to view the stable, simple and self-evidently given quality of computational objects as the outcome of a complex sociotechnical genesis.

Encapsulation, Exceptions and Unknowability

Thus far we have sought to address material aspects of the complex processes of abstraction that are at work in object-oriented programming. We need nevertheless to insist that computational objects do have a cognitive role. However, this is a role that is fulfilled primarily through the – often blind and groping – ways in which they give shape to non-computational processes.

The world in which computational objects operate is one to which they relate through precisely defined contractual interfaces that specify the interplay between their private inner workings and their public façades. One doesn't interact with a machine any old how but with a latitude for freedom, a room for manoeuvre, that is very precisely, programmatically, specified. *Encapsulation*, often held to be one of the primary features of object-oriented programming, enforces – along with the *exception* construct – a strict demarcation of inside and outside that is only bridged through the careful design of interfaces, making 'not-knowing' into a key design principle. The term 'encapsulation' refers to the way in which object-oriented programming languages facilitate the hiding of both the data that describes the state of the objects that make up a program, and the details of the operations that the object performs.[27] In order to access or modify the data descriptive of the state of an object one typically uses a 'get' or a 'set' 'method', rendering the nature of the interaction being accomplished explicitly visible. Encapsulation offers a variant, – at the level of the formal constructs of a programming language, – of a more general principle observed by programmers, which is that when writing an interface to some element of a program, one should always hide the 'implementation details', so that users do not know about and are not tempted to manipulate data critical to its functioning. 'User' here is a term relative to the objects under consideration: programmers don't allow users of websites to play around with the code that sets the terms on which a web browser operates, and one group of programmers may hide the implementation details of a set of computational objects from any other programmers who may want to use it, and so on.

In addition to promoting code re-use, it is said, encapsulation minimizes the risk of errors that might be created by incompetent programmers getting access to and manipulating data that might lead the object to behave in unexpected ways. A key maxim for programmers is that one should always code ‘defensively’, always write ‘secure’ code, and even accept that input – at whatever scale one wishes to define this – is always ‘evil’.[28] A highly regulated interplay between the inner workings and the outer functioning of objects makes it possible to ensure the stable operations of software. Arguably this is part of an historical tendency and proprietary trend to distance users from the inner workings of machines, effecting a complex sociotechnical knot of intellectual property, risk management and the division of labour,[29] the outcome of which is to restrict the programmer's ability to gain access to lower levels of operation (whilst theoretically making it easier to write code).

As a principle and as a technical constraint, encapsulation and the hiding of data at the very least gives shape to a technico-economic hierarchy in which the producers of programming languages can control the direction of innovation and change, by promoting ‘lock-in’ and structuring a division of work that encourages programmers to use proprietary class libraries rather than take the time to develop their own. By facilitating a particular – and now global – division of labour, the development of new forms of knowing through machines is in turn inhibited through the promotion of technically constrained, normative assumptions about what programming should be.

Whilst there are obviously many other factors that intervene to shape the way in which programming practices operate, the deeply sedimented habit of using class libraries is clearly something that has resulted from the technical affordances of encapsulation. A far more finely grained division of the work of software development is made possible when the system or application to be built can be divided into discrete ‘chunks’. Each class or class library (from which objects are derived) may be produced by a different programmer or group of programmers, with the details of the operations of the classes safely ignored by other teams working on the project. The contemporary trend towards the globalization of software development, with its delocalizing metrics for productivity, would not have acquired its present levels of intensity without the chunking of work that encapsulation facilitates.[30]

Finally, let us look briefly at *exception handling*. Where encapsulation works to create stabilized abstractions by closely regulating the interplay between the inside and the outside of computational objects, defining what objects can know of one another, exception handling shapes the way in which computational objects respond to anything that exceeds their expectations. A program and the objects that make it up are only ever operative within a specified set of parameters, defining the relations it can have with its environment and embodying assumptions that are made about what the program should expect to encounter within it. If those assumptions are not met (your browser is missing a plug-in, say, or you deleted a vital.dll file when removing an unwanted application), the program will not operate as expected. Exception handling provides a way to ensure that the flow of control through a program can be maintained despite the failure to fulfil expectations, ensuring that an application or system need not crash simply because some unforeseen problem has occurred. And in object-oriented programming, an exception is an object like any other: one can create subtypes of it, extend its functionality and so on.[31]

Technically, the rationale for exceptions is well understood and their treatment as objects, with everything that entails, facilitates their programmatic handling. What is less recognized, though, is the way that practices of exception handling give material shape to the kinds of relations computational objects have with the outside. From the point of view of computational objects, the world in general is a vast and largely unknown ensemble of events, *to* which such objects can only have access under highly restricted conditions, but also *in* which those objects only have a very limited range of interest – the role of the programmer being to specify this range of pertinences as precisely as possible. This is something that can be achieved in many ways: the practice of 'validating' user input, for example (by checking that the structure of that input conforms to some previously specified 'regular expression', say), ensures that the computational objects that process that input do not encounter any surprises that they won't understand (such as a date entered in the wrong format).

Because the use of exception handling in a program makes it possible for computational objects to continue to go about their work without there being too much disruption, and because their status as computational objects in their own right allows them to be programmatically worked with in the same way as other objects,[32] the need to pay closer attention to what causes the problems giving rise to the exceptions in the first place (systems analysis and design decisions, the framing of the specification of the software and so on) is minimized. The common practice of programmatically 'writing' information about the problems that give rise to exceptions to a log file – because this enables software developers to identify difficulties in program design, the routine causes of problems and so on – mitigates this ignorance to a point. However, it must be understood that the information thus derived presumes the terms in which the software defined the problem in the first place. As a result, one can only make conjectures about the underlying causes of that problem (a log file on a web application that repeatedly logs information indicating that a database server at another location is not responding cannot tell us if the server has been switched off or broken down, for example).

The point is that because exception handling facilitates the smooth running of software, it helps to stabilize not only the software itself but also the programming practices that gave rise to it. Exceptions work to preserve the framing of technical problems *as* technical problems, allowing errors to be defined, typically, as problems that the user creates through not understanding the software (rather than the other way round). In this way, exception handling obviates the need to develop a closer consideration of the relationship between computational objects and their environment. Whilst such stabilization allows software to gain a certain unobtrusiveness, this 'grey' quality makes it difficult to get a better sense of the differences that its abstract materiality produces.[33]

Conclusion: Ontological Modelling and the Matter of the Unknown

In the course of this chapter we have endeavoured to sketch out some of the reasons for developing an account of object-oriented programming which considers computation not from an epistemic but from an ontological point of view. It is true that there is an historically well-sedimented association between computation and discourses about knowledge, that computer programming seeks to model reality, that there are links between programming languages and formal logic, and so on. But this is not enough to make understanding computer programming as a science, in the way that, say, physics, chemistry or even the social sciences (sometimes) are understood, a legitimate move. On the contrary, this chapter has tried to suggest, through its examination of some of the features of object-oriented programming, that the abstractive capture and manipulation of agency through software in the calculus of computational objects are better understood as an ensemble of techniques engaged in a practice of *ontological modelling*. In other words, computer programming involves a creative working with the properties, capacities and tendencies offered to it by its environment which is obscurely productive of new kinds of entities, about which it may know very little. Such entities make up the fabric of what we have called here 'abstract materiality', a term we have used to gesture towards the consistency and autonomy of the zones or territories in which computational objects interface with other kinds of entity.

Despite the common connection that is made between computers and knowledge, a connection that is particularly evident in discussions of object-oriented programming and the modelling it accomplishes, a certain kind of unknowability is produced through the technically facilitated processes of abstraction that object orientation yields. This is not simply because encapsulation yields black boxes that parcel out and separate off knowledge or capacities for action – even amongst those that might be technically capable of knowing and working with the innards of such objects – but also because the epistemic valences of object orientation tend to obscure an aspect of computing most often associated with the circulation and refinement of meaning: that of the manipulation and evaluation of symbols. Object orientation draws from interpretation, via abstraction, in order to make things in the world. In this we can remember its genesis in two forms of constructivism: the psychologically derived approach of Kay and others at the Palo Alto Research Center; and that of the technosocial assembled in Scandinavia. That the capacity of objects finds itself in other kinds of alliance at other points, drawing out its acquired ability to hide, rather than to reveal relations, shows its exemplarily paradoxical double agency. Object orientation might be approached from many points of view: the angle taken here is one which insists, in a manner analogous to that of Michel Foucault discussing power, that the problem is not that the sociotechnical practice of programming doesn't know what it is doing. Rather, the techniques and technologies of object orientation produce a situation in which one doesn't know what one does.

References

Keld Bødker, Finn Kensing and Jesper Simonsen, *Participatory IT Design: Designing for Business and Workplace Realities*, MIT Press, Cambridge, MA, 2004.
Jacques Camatte, *LIP and the Self-Managed Counter-Revolution*, trans. Peter Rachleff and Alan Wallach, Black & Red, Detroit, 1975.
Edgar F. Codd, *The Relational Model for Database Management*, 2nd edn, Addison-Wesley, Reading, 1990.
Gilles Deleuze and Félix Guattari, *A Thousand Plateaus*, trans. Brian Massumi, University of Minnesota Press, Minneapolis, 1987.
Gilles Deleuze and Claire Parnet, *Dialogues*, Athlone, London, 1987.
Matthew Fuller and Andrew Goffey, *Evil Media*, MIT Press, Cambridge, MA, 2012.
Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Indianapolis, 1995.
Stephen J. Goldsack and Stuart J. H. Kent (eds.), *Formal Methods and Object Technology*, FACIT Series, Springer, Vienna, 1996.
Jan Rune Holmvik, 'Compiling SIMULA: A Historical Study of Technological Genesis', *IEEE Annals of the History of Computing*, 16:4 (1994), pp. 25–37.
Daniel Inglis, 'Design Principles Behind Smalltalk', *BYTE*, 6:8 (1981), pp. 286–302.
Alan Kay, 'The Early History of Smalltalk', http://gagne.homedns.org/%7etgagne/contrib/EarlyHistoryST.html
.
Alan Kay, 'Prototypes Versus Classes', *Squeak Developers' Mailing List*, 10 October 1998, http://lists.squeakfoundation.org/pipermail/squeak-dev/1998-October/017019.html

Rob Kitchin and Martin Dodge, *Code/Space. Software and Everyday Life*, MIT Press, Cambridge, MA, 2011.
Friedrich Kittler, 'Protected Mode', in Kittler, *Literature, Media, Information Systems*, ed. and trans. John Johnston, G&B Arts, Amsterdam, 1997.
Audris Mockus and David M. Weiss, 'Globalization by Chunking: A Quantitative Approach', *IEEE Software*, 18:2 (2001), pp. 30–7.
Kristen Nygaard and Ole-Johan Dahl, 'The Development of the SIMULA Languages', *ACM SIGPLAN Notices*, 13:8 (1978), pp. 245–72.
Anton Pannekoek, *Workers' Councils*, J. A. Dawson, Melbourne, 1950, https://www.marxists.org/archive/pannekoe/1947/workers-councils.htm
Andrew Pickering, *The Mangle of Practice: Time, Agency and Science*, University of Chicago Press, Chicago, 1995.
Steven Shaviro, *Without Criteria: Kant, Whitehead, Deleuze and Aesthetics*, MIT Press, Cambridge, MA, 2009.
Jaroslav Sklenar, 'Introduction to OOP in SIMULA', based on the 30 Years of Object Oriented Programming seminar, University of Malta, 1997, http://staff.um.edu.mt/jskl1/talk.html
Alfred Sohn-Rethel, *Intellectual and Manual Labour: A Critique of Epistemology*, Humanities Press, Atlantic Highlands, 1977.
Isabelle Stengers, *La vierge et le neutrino: les scientifiques dans la tourmente*, Empecheurs de Penser en Rond, Paris, 2006.
Isabelle Stengers, *Thinking with Whitehead: A Free and Wild Creation of Concepts*, trans. Michael Chase, Harvard University Press, Cambridge MA, 2011.
Bjarne Stroustrop, *A History of C++: 1979–1991*, AT&T Bell Laboratories, Murray Hill, n.d.
Alfred Tarski, 'Truth and Proof', *Scientific American*, June (1969); reprinted in R. I. G. Hughes (ed.), *A Philosophical Companion to First-Order Logic*, Hackett, Cambridge, MA, 1993.
Alberto Toscano, 'The Open Secret of Real Abstraction', *Rethinking Marxism*, 20:2 (2008), pp. 273–87.
Alfred North Whitehead, *Process and Reality*, ed. David Ray Griffin and Donald W. Sherburne, Macmillan, New York, 1979.

Notes

1 We borrow the notion of territorialization and its cognate terms – de-and reterritorialization – from Gilles Deleuze and Félix Guattari. They are discussed in further detail later.

2 Andrew Pickering, *The Mangle of Practice: Time, Agency and Science*, University of Chicago Press, Chicago, 1995.

3 Gilles Deleuze and Claire Parnet, *Dialogues*, Athlone, London, 1987, p. 52. Translation slightly modified.

4 Kristen Nygaard and Ole-Johan Dahl, 'The Development of the SIMULA Languages', *ACM SIGPLAN Notices*, 13:8 (1978), pp. 245–72; Jaroslav Sklenar, 'Introduction to OOP in SIMULA', based on the 30 Years of Object Oriented Programming seminar, University of Malta, 1997, http://staff.um.edu.mt/jskl1/talk.html; Jan Rune Holmvik, 'Compiling SIMULA: A Historical Study of Technological Genesis', *IEEE Annals of the History of Computing*, 16:4 (1994), pp. 25–37.

5 As in the work of Edgar Codd, for example. Edgar F. Codd, *The Relational Model for Database Management*, 2nd edn, Addison-Wesley, Reading, 1990.

6 A full description of workers' councils is not possible here. But for a historically precedent and advanced strand of work see that of Jacques Camatte and Anton Pannekoek. See Jacques Camatte, *LIP and the Self-Managed Counter-Revolution*, trans. Peter Rachleff and Alan Wallach, Black & Red, Detroit, 1975; Anton Pannekoek, *Workers' Councils*, J. A. Dawson, Melbourne, 1950, https://www.marxists.org/archive/pannekoe/1947/workers-councils.htm.

7 Operational research, or operations research in the USA, is a field arising out of wartime logistics, in which, for instance, the whole process of managing a supply or chain was thought of as including all levels of production and use, aiming to integrate them in ways that would maximize certain sorts of efficiency.

8 Participatory design itself is a changing domain, involving numerous conflicting tendencies, becoming in some cases simply one of a number of available techniques for effective knowledge

management and decision-making within a project and the systematic garnering of relevant human factors without any attempt to stage a critical analysis of work. At the same time, it remains as a persistent demand and exemplar for reasoned relations between the objects and processes of work and those that work with them. See, for instance, Keld Bødker, Finn Kensing and Jesper Simonsen, *Participatory IT Design: Designing for Business and Workplace Realities*, MIT Press, Cambridge, MA, 2004.

9 See Nygaard and Dahl, 'Development of the SIMULA Languages'.

10 In other words, to have a memory of the settings of all their variables. Imagine an object called 'bank account' which kept on forgetting things like 'bank balance', 'account holder' *etc.*

11 See Bjarne Stroustrop, *A History of C++: 1979–1991*, AT&T Bell Laboratories, Murray Hill, n.d.

12 Kay is a researcher with an intellectual trajectory stretching back to Ivan Sutherland and the invention of Sketchpad, the first CAD (computer-aided design) machine, a system that itself had an understanding of a certain kind of digital object at its core. One only has to survey contemporary computing culture to see the extent of the influence, by more or less direct means of the work groups of which he was part. See, for a contemporary overview of Smalltalk the special issue of *BYTE*, 6:8 (1981).

13 Alan Kay, 'Prototypes Versus Classes', *Squeak Developers' Mailing List*, 10 October 1998, http://lists.squeakfoundation.org/pipermail/squeak-dev/1998-October/017019.html.

14 See Daniel Inglis, 'Design Principles behind Smalltalk', *BYTE*, 6:8 (1981), pp. 286–302.

15 An interesting comparison is to the work of Seymour Papert, developer of the LOGO programming language designed specifically for education, in that both languages were designed with an explicit attention to their epistemic consequences. Both

languages, in various different incarnations, are maintained by active user communities.

16 Alan Kay, 'The Early History of Smalltalk', http://gagne.homedns.org/%7etgagne/contrib/EarlyHistoryST.html.

17 Kay 'The Early History of Smalltalk'.

18 On this point, Kay invokes Carnap, whose 1947 essay *Meaning and Necessity* offered a sophisticated development of the notions of intension and extension in logic.

19 A philosophical reference might make this clearer. The notion of synthesis we are pointing to should not be understood along the lines of the syntheses outlined by Kant in his *Critique of Pure Reason*, which still revolve around representation and place a human subject at their centre. The relevant reference instead is Whitehead, whose prehensions offer a principle of synthesis that doesn't depend on representational presuppositions, and allow subjects to 'emerge' as part of the 'concrescent' process of which any prehension is a part. See the third part of Alfred North Whitehead, *Process and Reality*, on the theory of prehensions. Technically, the computational capture of agency involves both prehension *and* ingression, in the sense that a computational object gives a (coded) form of determinateness to whatever it captures. Whiteheadian ingression is not unlike 'recording' in the early work of Deleuze and Guattari. See Alfred North Whitehead, *Process and Reality*, ed. David Ray Griffin and Donald W. Sherburne, Macmillan, New York, 1979. See also Steven Shaviro, *Without Criteria: Kant, Whitehead, Deleuze and Aesthetics*, MIT Press, Cambridge, MA, 2009.

20 Such modelling might be qualified as ontological here because of the way that it works on the mode of existence of users, giving shape to their habits, their expectations and anticipations. This is something that is difficult to 'see' if thought in terms of an epistemological problematic of representation. We return to this question in the conclusion.

21 To borrow from biology, we might say that that history is both onto-and phylo-genetic.

22 The notion of *real abstraction* is one associated with Marx and Marxism. See in particular Alfred Sohn-Rethel's work on intellectual and manual labour and the discussion of it by Alberto Toscano. Alfred Sohn-Rethel, *Intellectual and Manual Labour: A Critique of Epistemology*, Humanities Press, Atlantic Highlands, 1977; Alberto Toscano, 'The Open Secret of Real Abstraction', *Rethinking Marxism*, 20:2 (2008), pp. 273–87. Whitehead and Deleuze and Guattari enable us to develop a precise understanding of this process.

23 Gilles Deleuze and Félix Guattari, *A Thousand Plateaus*, trans. Brian Massumi, University of Minnesota Press, Minneapolis, 1987, p. 193.

24 Christopher Alexander, quoted in Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Indianapolis, 1995, p. 2.

25 Stephen J. Goldsack and Stuart J. H. Kent (eds.), *Formal Methods and Object Technology*, FACIT Series, Springer, Vienna, 1996.

26 This is a theme explored in Isabelle Stengers, *Thinking with Whitehead: A Free and Wild Creation of Concepts*, trans. Michael Chase, Harvard University Press, Cambridge MA, 2011. The 'patience' of the environment is what allows the infectious dynamics of power to operate effectively.

27 Not all computer scientists or software engineers agree that encapsulation is the same thing as information or data hiding. The details of the disagreement need not concern us here.

28 See Alfred Tarski, 'Truth and Proof', *Scientific American*, June (1969); reprinted in R. I. G. Hughes (ed.), *A Philosophical Companion to First-Order Logic*, Hackett, Cambridge, MA, 1993. For further discussion see Matthew Fuller and Andrew Goffey, *Evil Media*, MIT Press, Cambridge, MA, 2012.

29 This is one way in which to read the rather deterministic argument proposed by Friedrich Kittler in his essay 'Protected Mode', in Kittler, *Literature, Media, Information Systems*, ed. and trans. John Johnston, G&B Arts, Amsterdam, 1997.

30 The global division of programming labour is discussed in Audris Mockus and David M. Weiss, 'Globalization by Chunking: A Quantitative Approach', *IEEE Software*, 18:2 (2001), pp. 30–7. Notwithstanding the tensions implied by the technosocial genesis of the object, in this present context, it is a subject worth a study in its own right. It is particularly instructive in this regard to compare the distribution and use of code modules for the scripting language Perl with the proprietary class libraries associated with a language like Microsoft's C#. It is also worth noting that one of the criticisms made of object-oriented programming is that it doesn't require any skill to program in, a criticism that can clearly be read in the light of the trend pointed towards here.

31 One might, for example, refer to Microsoft's documentation of the System. Exception class for details of the complex structure of inheritance relations, the properties and methods of exception objects in the C# language, its subclasses and so on.

32 See our earlier comments on inheritance.

33 That techniques and technologies of software culture are bound up in a logic of 'the same' which precludes a deeper understanding of the transformations that they accomplish here finds some elements of an explanation. On the logic of the same and technology, see Isabelle Stengers, *La vierge et le neutrino: les scientifiques dans la tourmente*, Empecheurs de Penser en Rond, Paris, 2006. On the unobtrusiveness of software, see Rob Kitchin and Martin Dodge, *Code/Space. Software and Everyday Life*, MIT Press, Cambridge, MA, 2011.