19

The unknown objects of object-orientation

Matthew Fuller and Andrew Goffey

Introduction

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 incorporated in a strictly defined ordering of routines and subroutines. 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 be a computational object, or must be a property of a computational object. Object-oriented programming is typically distinguished from earlier procedural (such as C) and functional (such as Lisp) programming, declarative programming (Prolog) and, currently, component-based programming. Some of today's most widely used programming languages (Java, C#) have a decidedly object-oriented flavour, and object-orientation 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 chapter 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). Developing a consideration of computational objects that goes beyond their technoscientific enframing in representational terms (the idea that computational objects are models of real-world entities) the chapter asks more broadly what powers computational objects have, what effects they produce and, more importantly perhaps, how they produce them. Our aim is to make perceptible the territorializing powers of computational objects, as they work to model and remodel relations, processes and practices in a paradoxically abstract material space. Such powers might be best understood in terms of a process of *ontological modelling*. Computational objects are susceptible of a notation that frames them as *epistemic operators*, the technoscientific incarnation of concepts in a formal-logical calculus. However, framing them in these terms by no means exhausts their pragmatic virtues, their material efficacy, which bears more enduringly and obscurely on the new forms of existence that they bring into play, shaping and reshaping users, their dispositions and habits.

Whereas in its 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, the formal qualities of computation might appear divorced from the rich material textures of culture and a concern with the ontological dimension of 'things', computation also has very efficacious and productive traction 'in the real'. The formal calculus of signs that permits the effective resolution of a problem of computation is at the same time a matter of the successful creation, through programming, of a more or less stable set of material processes, within, but also without, the casing of the machine. We refer to calculus of signs as 'abstract materiality', because computational objects are not really immaterial; their operations unfold on a material plane, a plane of relative 'consistency' composed of specific forms of agency that are abstracted from other kinds of material processes and composed in a sometimes fragile ecology of relations. At the same time, computational objects remain in contact with the material processes from which they are abstracted but only through 'redefining' or pacifying them, making them invisible and/or problematic.

To take up the question of the production of this abstract materiality and its unsettling qualities, we briefly consider object-oriented programming and its transformative effects, addressing object-orientation as a sociotechnical practice. The argument develops in a number of stages. A first section considers several key features of object-oriented programming through a consideration of its earliest avatars, and highlights an ambiguity in understandings of programming as a kind of modelling, the implications of which we then start to unpack in the second section. Here, we seek to address such computational objects in non-representational terms as a set of processes capturing agency, allowing us to reframe programming in sociotechnical terms that do not fall into the trap of representation. We then turn our attention to a more direct consideration of the efficacy of programming as a constructive process of modelling with objects. Of particular importance is the question of how typical programming constructs, such as design patterns, operate to stabilize patterns of relations between objects and their environments. In a final section, we explore in more detail some of the ways in which the programming practices that develop out of the use of computational objects are generative of obscurity, unknowability and ignorance. Two main arguments are developed: (1) computational objects have historically been understood in technoscientific terms as a set of formal-material 'concepts', and yet (2) in the sociotechnical qualities of their development and deployment they concomitantly operate as a limit *to* and as limiting *of* knowledge. In this respect, computational objects form a critical relay in the generation of relations of power, which thus become something that is exercised in the supple fabric of materiality that they generate.

Modelling objects

Object-oriented programming comes into the world through the development of new forms of programming language. Such languages are 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. Every programming language forms a carefully and precisely constructed set of protocols established in view of historically, technically and organizationally specific problems. This is as true in the case of object-oriented languages as it is with other kinds of programming language. For instance, the SIMULA language, developed in Norway in the 1960s, aimed at providing a means to both describe (that is, program) a flow of work, and to simulate it, with the aim of bringing the capacity to design work systems (despite their relative technical complexity) into the purview of those who made up a workplace. In this respect, 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, and developed out of a tradition that would become known as 'participatory design' (Bødker *et al.* 2004). The first version of SIMULA, SIMULA I, was not developed with a view to establishing object-orientation as a new format for programming languages per se but rather as a way of modelling the operation of complex systems. Although it was not greatly popular as a general programming language, SIMULA's technical innovation of providing for structured blocks of code (called classes), which would eventually be instantiated as objects, was taken up fifteen years later in the development of C++, a language developed in part to deal with running UNIX-based computational processes across networks (Stroustrop n.d.), which later became a driver in the development of another object-oriented programming language, Java.

The crucial feature of object-oriented programming as a way of modelling entities in software becomes more obvious in another precursor of today's programming languages. In the language Smalltalk, developed under the leadership of Alan Kay at Xerox PARC, in the 1970s and 1980s, it is the relations *between* things that become central (Kay 1998). Objects are generated as instances of ideal types or classes, but their actual behaviour is something that arises from the messages passed to them 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. This is important because although the effective order in which computational messages/events are executed is essentially linear, the order itself is not rigidly prescribed and there is an overall sense of polyphony of events and entities in dynamic relation. This represents a significant shift in relation 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 flexible relationship between the user and the machine.

These early developments in object-oriented programming languages both point towards and beyond the epistemic framing of computational objects and the ambivalence that computer programming as a kind of modelling incarnates. Kay (1998), for example, argues that '… 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'. However, if that were the case, modelling could not be understood as a simple representation. This is because when the complex dynamic systems and user relationships that object-oriented design models are those that are made possible by technology, it is no longer a matter of representing the world through artefacts but of creating models of non-preexistent things. Yet, the framing of computational objects in epistemological terms as representative concepts is widespread. Kay (1993) 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'. On this count, a computer program written in an object-oriented programming language develops a kind of intensional logic, in the sense that the objects that the program comprises each have an internal conceptual structure that determines its relationship to what it 'refers' to. Computational objects are, in this sense, concepts that offer a mechanistic materialized representation of objects generated through a kind of logical calculus. In this respect, we can consider that computational objects have an 'analytic' function, embodying an understanding of the entities that they model. However, given that the 'silicon explosion' makes possible new kinds of system and new kinds of relations, a computational object must be seen as having a 'synthetic' function that adds to and is in excess of the reality it might otherwise be thought to model. It is, we would suggest, this second, synthetic aspect of computational objects (in which modelling is not so much modelling *of* objects, but modelling *with* objects) that needs to be understood more precisely.

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 having 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 computers and humans. Indeed, given what we have said about modelling, it would be more appropriate to consider these relations in terms of a series of forms of the abstractive creation, *capture* and codification of agency: a click of the mouse, a tap of the key, data input, affective investments and so on. By referring to the creation and capture of agency we seek to underline two things: (1) computational objects do not simply or straightforwardly build on preformed capacities or abilities, they generate new kinds of agency, which may be similar to what went before but are nevertheless different (for example, a typewriter, a keyboard and a keypad capture the agency of fingers in subtly different ways); (2) the agency that is created is part of an asymmetric relation between human and computer, a kind of cultivation or inculcation of a *mechanic habitus*, a set of dispositions that is inseparable from the technologies that codify it and give it expression (Pickering 1995). It would be too long a diversion to examine in detail how and why this asymmetry exists, but such asymmetry 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 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, open a web page or even closing down altogether. 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. In this sense, 'bugs' have played an important role in setting up the asymmetric relations between humans and machines. Relatively speaking, humans adapt to (or at least learn not to notice) the stupidities of the computer more quickly than the computer adapts to humans, in part simply because the time between software releases (with bug fixes) is 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 and a dual meaning in *modelling* the user with which they interact.

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 with which it can be interacted. For example, the objects (text boxes, lists, hyperlinks and so on) that populate a web page define more or less exactly what a user can do. This is a function not just of how the site has been designed and built but, importantly, of a range of previously defined sets of coded functionality. (A website is dependent on a browser, which is in turn dependent on the operating system of the machine on which it operates). There is a history to each of these objects and their development, which means that the parameters for interaction are determined by a series of more or less successful *abstractions* of a peculiarly composite, multilayered and stratified kind.

However, it is important to note here that abstraction is a contingent, *real*, process. The taken-for-granted ways in which humans now interact with machines are the product of material

arrangements that do not always apply. More pointedly, these real abstractions are concretely and endlessly reactualized 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 (1987) sense. 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 us understand this dynamic?

One feature that is associated in particular with object-oriented programming is the way that it is argued to facilitate the *reuse* 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 reuse it in different programs. Reusability is not unique to object-oriented programming; the routinization and automation of computational tasks implies it as a basic operating feature of software per se. But object-oriented programming favours the reusability 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 to 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 uses 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 reuse 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 their ecological *niches*.

The possibility of reusability 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. The simple dynamics of adaptation or habituation, we have just suggested, might account for the latter. The former can be located directly in the technical features of object-oriented programming languages. We address reusability 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 denoting a relationship: a Persian or a Siamese *is a* cat, a savings account *is an* account. 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 but crucially 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 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 broadly accepted 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 offers a discrete way of capturing practices and relations in software through a logic of imitation (cf. Tarde in Deleuze and Guattari 1987).

Design patterns extend this logic of code reusability to the situation of a more complex set of algorithms designed to address a broader problem. A design pattern in software provides a reusable solution to the problem posed by a computational context, and although 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 (Gamma *et al.* 1994; Shalloway and Trott 2005). This is what makes it interesting, because its very existence is evidence of the increasing complexity and shifting social relations 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 creates design problems entailing a different set of object-relations than in 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 different things. Design patterns imply varied sets of relations between software and users, perhaps entailing an automated set of links between one company and companies further up the supply chain. The latter might thus reasonably be expected to entail online 'business-to-business' communication on a 'multitier' model, whereas the former might adopt a more traditional 'client–server' relationship. Although a user might experience these as similar, the relations between the objects that make them up are considerably different.

In one respect, such patterns respond to the core difficulty of object-oriented software development: the analytic decomposition 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. Their very existence is interesting not just because they provide evidence 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 thinking in object-oriented terms is generally shored up in textbooks by means of analogy with more commonsensically object-like objects: tables, chairs, papers, books, and so on (Goldsack and Kent 1996). Yet, the stability of an environment is absolutely critical in enabling computational objects to exert their powers effectively (Stengers 2011). However, the pedagogical emphasis on the relatively simple does not do justice to the processes at work in the historical development of software culture. Perhaps, it would be more appropriate 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. This role is fulfilled primarily through the, often blind and groping, ways in which such objects 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 public façades. One does not interact with a machine any old how but with a latitude for freedom that is precisely, programmatically, specified. Along with the *exception* construct, *encapsulation* is often held to be one of the primary features that object-oriented programming enforces, 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 describe the state of the objects that make up a program, and the details of the operations that the object performs.[1] 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.

In addition to promoting code reuse, 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' (Howard and LeBlanc 2003). 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 a 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, 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 give shape to a technicoeconomic 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. Indeed, a 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.[2]

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 does 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[3]), suggesting it too facilitates the logic of imitation.

Technically, the rationale for exceptions is well understood and their treatment as objects, with everything that entails, facilitates their programmatic handling. What is less well understood, however, is the way that practices of exception handling give material shape to the kinds of relations that computational objects have with their 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 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 processing that input do not encounter any surprises (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 go about their work without 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, 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 such 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 ever 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), leaving the commonplace of the information technology helpdesk (for users having problems with software, 'read the fucking manual') as evidence of the structure of judgement this situation yields.

The point is that because exception handling facilitates the smooth running of software, it not only helps to stabilize 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 typically defined as problems that the user creates through not understanding the software (rather than the other way round). In this way, exception handling obviates developing a closer consideration of the relationship between computational objects and their environment

or problematizing the framing of programming practices. Although this can allow software to gain a certain routinized unobtrusiveness (Kitchin and Dodge 2011), this 'grey' quality makes it difficult to obtain a better sense of the differences (Stengers 2006) that its abstract materiality produces.

Conclusion: ontological modelling and the matter of the unknown

In the course of this chapter, we have endeavoured to sketch out some reasons for developing an account of object-oriented programming that considers computation not from an epistemic but from an ontological point of view. It is true that there is 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, a legitimate move. On the contrary, we have tried to suggest that the abstractive capture and manipulation of agency through software in the calculus of computational objects is 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 that is obscurely productive of new kinds of entities, about which it may know very little. Such entities make up the fabric of 'abstract materiality', a term that gestures towards the consistency and autonomy of the zones or territories in which computational objects interface with other kinds of entity.

Object-orientation might be approached from many points of view: the angle taken here is one that insists, in a manner analogous to Michel Foucault discussing power, that the problem is not that the sociotechnical practice of programming does not know what it is doing. Rather the techniques and technologies of object-orientation produce a situation in which one does not know what one does does.

Notes

1 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.
2 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. The global division of programming labour is discussed in Mockus and Weiss 2001 and Greenspan 2005.
3 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.

Bibliography

Bødker K., Kensing F. and Simonsen, J. (2004) *Participatory IT Design, Designing for Business and Workplace Realities*, Cambridge, MA: MIT Press.
Deleuze G. and Guattari F. (1987) *A Thousand Plateaus*, trans. Brian Massumi, Minneapolis: University of Minnesota Press, p. 193.
Gamma E., Helm R., Johnson R. and Vlissides J. (1994) *Design Patterns. Elements of Reusable Object-Oriented Software*, Indianapolis: Addison-Wesley.
Goldsack S. and Kent S. (eds) (1996) *Formal Methods and Object Technology*, New York: Springer-Verlag.
Greenspan A. (2005) *India and the IT Revolution, Networks of Global Culture*, London: Palgrave Macmillan.
Howard M. and LeBlanc D. (2003) *Writing Secure Code*, Redmond WA: Microsoft Press.

Kay A. (1993) *The Early History of Smalltalk*, http://www.smalltalk.org/smalltalk/TheEarlyHistoryOfSmalltalk_Abstract.html/

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

Kitchin R. and Dodge, M. (2011) *Code/Space. Software and Everyday Life*, Cambridge, MA: MIT Press.

Mockus A. and Weiss D.M. (2001) 'Globalization by Chunking, a Quantitative Approach', *IEEE Software*, March/April, pp. 30–37.

Pickering A. (1995) *The Mangle of Practice*, 2nd edition, Chicago: Chicago University Press.

Shalloway A. and Trott J.R. (2005) *Design Patterns Explained: A New Perspective on Object-Oriented Design* (2nd Edition), Boston: Addison-Wesley.

Stengers I. (2006) *La vierge et le neutrino*, Paris: Les empêcheurs de penser en rond.

Stengers I. (2011) *Thinking with Whitehead, a Free and Wild Creation of Concepts*, trans. Michael Chase, Cambridge, MA: Harvard University Press.

Stroustrop B. (2007) *A History of C++: 1979–1991*, www2.research.att.com/~bs/hopl2.pdf/

Objects and Materials

A Routledge Companion

Edited by Penny Harvey, Eleanor Conlin Casella, Gillian Evans, Hannah Knox, Christine McLean, Elizabeth B. Silva, Nicholas Thoburn and Kath Woodward

Routledge
Taylor & Francis Group
LONDON AND NEW YORK

First published 2014
by Routledge
2 Park Square, Milton Park, Abingdon, Oxon OX14 4RN

Simultaneously published in the USA and Canada
by Routledge
711 Third Avenue, New York, NY 10017

Routledge is an imprint of the Taylor & Francis Group, an informa business

British Library Cataloguing in Publication Data
A catalogue record for this book is available from the British Library

Library of Congress Cataloging in Publication Data
Objects and materials: a Routledge companion / edited by Penny Harvey, Eleanor Conlin Casella, Gillian Evans, Hannah Knox, Christine McLean, Elizabeth B. Silva, Nicholas Thoburn and Kath Woodward.
pages cm
Includes bibliographical references and index.
1. Material culture. 2. Ceremonial objects. 3. Art objects. I. Harvey, Penelope,
GN406.O28 2013
930.1–dc23 2012049615

ISBN: 978-0-415-67880-3 (hbk)
ISBN: 978-0-203-09361-0 (ebk)

Typeset in Bembo
by Cenveo Publisher Services