Law Text Culture

Volume 23 *Legal Materiality* Article 8

2019

Against the Power of Algorithms Closing, Literate Programming, and Source Code Critique

Markus Krajewski
University of Basel, Switzerland

Against the Power of Algorithms Closing, Literate Programming, and Source Code Critique

Abstract

Current commentaries on digital change have emphasised the reality of our increasing exposure to the power of algorithms. My intervention examines this assertion with a media-historical approach that traces arguments raised in a legal case to the point of software inception. I show that the power of algorithm is based on the eminent cultural techniques of reading and writing. As an antidote to this power, I propose the concept of source code critique, which draws upon historiography and so-called 'literate programming', and which could help to introduce transparency into an algorithm's opaque agency.

Against the Power of Algorithms
Closing, Literate Programming, and Source Code Critique

Markus Krajewski

1 Finnish, White, Male, Rural Resident Putting an Algorithm on Trial

Some algorithms know more than we think they would know at certain moments: the notorious examples range from superior, hidden knowledge to explicit oppression. A famous example is one of a large American retail chain that knew of a teenage customer's pregnancy long before her own father because of the goods in her online shopping cart (Mayer-Schönberger et al 2013: 57-58). In the context of so-called predictive policing, authorities pretend to know the next crime scene even before the crime is committed (Fry 2018: 144 ff.). And finally, criticism of Google's Page Rank algorithm continues: as Safiya Noble observes in her book *Algorithms of Oppression*, its design is based on an intricate series of racist assumptions. In the juridical realm, algorithms that technically consist merely of decisions between 0 and 1 on a basal level appear to play an increasing role in legal decision making. Sometimes algorithms themselves become 'defendants', as the following example from rural, sparsely populated Central Finland will show.

In early July 2015, a man in Central Finland put some building materials into an online shopping cart. Normally the last step of purchase is concluded on the basis of a credit contract. This process is easily activated with a few clicks, especially when a small amount is involved, as was the case here. To his surprise, however, the customer was informed that the credit company involved in the purchase contract had rejected the transaction. After unsuccessful and repeated attempts to find out the reason for the rejection and receiving unsatisfactory explanations, the man decided to take Svea Economi AB, Filial i Finland, the credit monitoring company, to court. The only explanation that Svea gave was that the denial of purchase had been based on a *credit score*, but it did not provide any further information about the score's exact function. Svea also revealed that they operated an algorithm that makes statistical decisions based on certain characteristics of the potential buyers, but that the algorithm did not take into account a consumer's financial solvency. Seeking more clarity about the algorithm and details about its function, the rejected customer decided to sue Svea.

The court case revealed the following: the credit monitoring service's assessment criteria were based on certain characteristics, such as the consumer's place of residence, gender, mother tongue and age. On the bases of population statistics and microcensus data, the algorithm then calculated the proportion of people in each group that had an unfavourable credit entry. Following a points system, it assigned a score to the individual consumer in that group. These points were then used to evaluate the probability of the consumer's ability to service a credit. The more points the consumer 'scored', the more creditworthy he or she was deemed to be. Men received fewer points than women; people with Finnish as their mother tongue received fewer points than those with Swedish mother tongues; and the region in Central Finland assigned a poorer reputation to middle-aged, male Finnish speakers in terms of payment morale than to their male peers in the urban parts of Finland. It was on the basis of these categorisations that the plaintiff seemed to have been denied credit.

The case reached the National Non-Discrimination and Equality

Tribunal of Finland. After two years of deliberation, the Tribunal decided to fine the defendant company, Svea, €100,000. It reasoned that the customer had not been treated as an individual in his creditworthiness, but only as a representative of a statistical profile based on discriminatory variables that Svea had applied to him. The defendant had been matched to all persons that fit his profile, i.e. men living in a given residential area, having a certain mother tongue and being of a certain age. Since the plaintiff fell into more than one category, the jury found that he was subjected to multiple discriminations. As a consequence of the Tribunal's findings, Svea announced that it would no longer use 'mother tongue' as a criterion for assessing creditworthiness.[1]

The case revealed the algorithm as an independent, justiciable actor whose actions were not only confined to 'inside' the computer (in a program sequence full of bugs or crashes), but which also acted outside the machine in the world, giving rise to legal effects that were 'faulty': it was acting illegally. Although technically speaking the algorithm was functioning flawlessly, the Finnish tribunal prohibited its continued use: the algorithm was at fault, both legally and socially, because the software design had been based upon discriminatory premises.

This case illustrates the opaqueness of algorithms' operations: a customer is left in the dark about the criteria through which a credit decision is made; and only court decisions can enforce a disclosure of a proprietary software's algorithm. Much of the conjured power of algorithms is thus based on the opacity or inscrutability of the specific actions that a software gradually applies and accrues. In the following sections, I outline the internal juridical character of algorithms, which resembles a mode of reasoning based on precedents: decisions are based on past decisions. I will then suggest some remedies to counter this power of algorithms.

2 Algorithms and Operational Chains: Social Constructs

There could hardly be anything less material in the physical sense of the term than an algorithm. Normally it has no gravity and is feather-light. It consists of rules, or more precisely, a set of commands executed by a body.[2]

In cultural history, algorithms have denoted a rule of calculation. Since the beginning of the High Middle Ages, calculus had replaced the prevalent practices of occidental calculating with technical devices, such as the abacus or the blackboard. Calculus favoured a sequence of instructions for action that would only allow scheduled mathematical operations to be carried out and posited an imaginary system of place values. But above all it proceeded *without further technical aids, media or objects*. The mathematician, geographer and astronomer, Abu Abdallah Muhammed ibn Musa al-Hwarizmi al-Magusi (approx. 780-850), often described as the founder of algebra, characterised the eponym of this procedure as al-Hwarizmi (= algoritmi): an algorithm denoted an immaterial instruction for action that involved an arithmetic operation consisting of a linear sequence of commands as its basis.[3] An algorithm came to be regarded as a self-contained chain of commands or a chain of operations. This is commonly referred to as a *code*.

Algorithms, however, could also be considered differently: in order to have an effect, a code requires a computer to execute it. Arguably no technique can only work on its own without the involvement of others. For this very reason it is worth emphasizing again that algorithms are social constructs. Their functions are always located at the interfaces between machines and humans. Not only do they process human inputs using data technology, but also in the vast majority of cases, software developers – i.e., humans – design the concrete algorithmic processes. At the same time, algorithms model their users, such as the Finnish borrower whose creditworthiness was modelled as the sum of points arising from individual 'properties', such as age, mother tongue, and so on.

The anthropologist André Leroi-Gourhan described algorithms as chains of operations, *chaîne opératoire* (1965/1980: 150 f., 275-280, 323).

In such chains, different actors are combined into a temporary unit of action that is only effective when objects, media and persons interact. An action only occurs if a chain of operations organises the interaction of a manual gesture (technique), a tool (*l'outil*) and symbolic operations (language/code) and thereby initiates it.[4] According to Latour's actor-network theory, chains of operations not only consist of technology and process, but are social constructs that have a social effect. They constitute and initiate the interaction of human and non-human actors (Latour 1991/2006, 2002). This characterisation of how a chain of operation is made and what it does applies well to algorithms: as a chain of operation in action, the execution of an algorithm generates not only virtual and social but also *physical* material effects; or as the case of the Finnish credit applicant has shown, it can also suppress them. In such an anthropological perspective, algorithms are chains of operations that transform certain ideas into concrete action by using tools or gestures.

Lastly, algorithms also exert a legal effect. An algorithm is not only an object of a tribunal or a court case; it can act as a legal material itself. As the Finnish case demonstrates, algorithms 'mediate and transform matters into distinctly legal matters' (Kang and Kendall, Introduction: 6). Moreover, algorithms themselves are also 'legal' in their core. Their operation resembles a law-making process in which their focal operation and fundamental mediational practice is one of *closing*. The next section will discuss the process of closing in more detail.

3 The Power of Closing Operations

Why are algorithms so powerful? They appear oblique or undecipherable not only because they operate opaquely and it takes a court case to open the source code. Rather, their power stems from the material form of the code itself. A code is a medium whose core functionality requires a closure.

Algorithms seem indiscernible and unintelligible because they are most commonly distributed to users as sealed, black-boxed packages. A single icon often serves as the gate to the software's interface.

Algorithms perform most of their tasks hidden behind neatly designed graphical surfaces. Of course there are plausible reasons for this intended invisibility. Parts of the code may include trade secrets, commercially valuable routines, or they are bounded by intellectual property rights. The international software industry has an interest in keeping these knowledges indiscernible and has no reason to disclose the sources of their commercial products. However, alternatives do exist for many of the commercially distributed software packages, and they provide more or less the same functionalities (for a sample, one may wish to try alternativeto.net in order to find free substitutes for commonly used commercial software). Such alternatives often are open-source software projects and they can be easily accessed on large repositories like github.com. Open software packages obtained in this way are copied (or cloned, as the proper term has it) to the user's local computer and accessed via an appropriate working environment, such as an IDE, which will be explained later in this text.

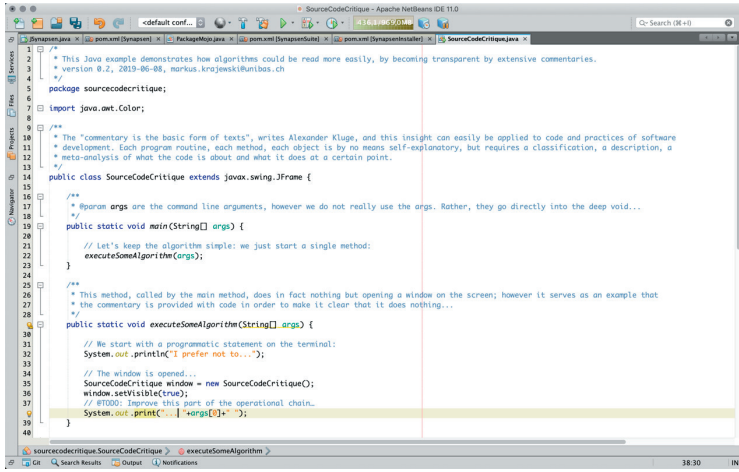

Figure 0: Code and comment coexist in the source code

For a long time, computer codes have been produced in an open

process in which software developers have deliberately written the codes and commented on them at the same time. Without being subjected or accountable to a single authority, such a practice has been marked by a collaboration of developers who have mutually commented on each others' codes. This was not only done in large scale projects. Software developers habitually comment – especially on their own algorithms – in order to define a code's current state (such as '@TODO: Improve this part of the operational chain…'), as well as to add personal notes (see fig. 0). Such commentaries may be necessary to understand the developer's own code, because the function of a code fragment or the manner in which it operates is no longer self-explanatory after a time. This is because the communication between the developer and the computer is mediated and entails multiple steps. Human software developers work on their source code with a set of commands provided by a higher computer language, such as Java or C#. These languages are quite abstract, but still form a readable and comprehensible sequence of words (see figure 0). The source code, cannot yet be understood by computers, however; it is not yet executable by the machine and requires an intermediate step to transform these algorithms into a machine-readable binary code. Such a transformation is usually the result of an interpretive process performed by the so-called compiler program. Source code therefore constitutes a wholly new genre of textual code (or coded text): constituting a hybrid, it can be regarded either as a collection of sequential commands or operational chains (which the machine will perform during runtime) or, in conjunction with the developer's commentaries, as a complete documentation of the task the machine is supposed to do if the process is initialized. The source code – literally both as a source and a code – contains the source of its own documentation, as well as the pre-form of the binary code.

The overlap and correlation between code and commentary in this peculiar state results in the highest level of a code's information density. Sources can be processed further with the help of specific utility programmes in an Integrated Development Environment (IDE, see fig. 1) in two ways: either by conversion into an executable file for machine use or by conversion into a comprehensible documentation

of the whole code, including commentaries and algorithms, which in effect represents a commentary on the process of codification itself. Both the algorithms and the commentaries evolve, whilst being deeply interwoven within the same file in the source code and distinguished only by certain tags and suffixes that identify their status.

These processes of compiling and/or documenting resemble a closing of codes; in other words, a *codification*. Such a closure of a code inserts a kind of juridical structure at the level of software development. While compiling input files, i.e. interpreting the commands line by line, the compiler freezes the code at a certain stage, and it becomes inalterable. It is at this point that the compiler translates the human readable chain of commands into a binary code (0 and 1) that can then be executed by the machine. These codes might contain orders about routines provided by the computer language itself (for example, opening a window on the screen or issuing a message, such as 'I prefer not to…' in the terminal window are performed). Tools such as compilers or pre-processors for documentation function as filters of the source code; they are not dissimilar to book editors. The compiler ignores the material that serves other purposes (for example, comments) and only selects those particular items necessary for the machine to process; while, vice versa, the documentation pre-processing tool uses the commentaries as the main text which are then illustrated by the commands.

The usage and meaning of the term 'code' are, of course, not restricted to computer coding. They operate in a wider realm of para-legality. Historically, the term 'code' has been deeply rooted in legal history, particularly in civil law jurisdictions (the Code Napoleon, codes of conduct, etc.). *Code* sounds somehow less determinate than *statute* or *das Gesetz*; codes formulate claims to something different, perhaps less sovereign, than state-enforced Law.

It is in this sense that computer codes have a juridical character, even if they derive from an open-source software culture of commenting and commentaries. In legal history, it is well known that the culture of commenting is not antithetical to codification and juridification. The opposite is rather the case: legal codes often derive from nothing other

than comments and commentaries. The famous *Digesta* that laid the ground for Western law drew its material from Roman jurists who had communicated to each other by commenting on cases. The large amount of these commentaries was eventually 'digested' and compiled in a book - a *codex* - from which the word 'code' derives. The Byzantine Emperor Justinian appointed an editor, Tribonian, who guided the process of converting comments into a single code by selecting certain data from the mass of text. Through this procedure of forming a book out of the abundance of commentaries, the *Digesta* established an inalterable legal text, as well as an ever-changing commentary to accompany it. The text which found its way into a codex between the two covers of a book literally becomes closed from that moment of entry; similarly, a compiler closes the source code. It can no longer be altered and thus ends late antiquity's non-hierarchical form of legal text generation and its incessant chains of comments without an ultimate reference. Such a closure of codes – later called codification – ends the practice of codes that appear and disappear according to their use.

4 Becoming Sovereign (Again): Source Code Critique

My initial question was: how could one overcome the issue of a code's opaqueness on a practical level?[5] What are the media and means that could be mobilised towards this aim? My suggestion may sound speculative, yet it addresses a code's inherent roots: I would like to propose a methodological framework following the idea of a *source code critique*. The term is a *mot-de-valise* from 'source code' and the helpful scholarly 'source criticism' proven in historiography (Saxer 2014: 376 ff.). Conceptually it involves, on the one hand, the field of source code development; on the other hand, it requires a critical reading of codes, the dynamics and changeability of which are to be understood as historical sources that require further classification and commentary.

I conceive source code critique less as a new research direction, as it was proposed about a decade ago by the so-called *Critical Code Studies* as a programmatic separation from the so-called *Software Studies* in order to study code as a text through hermeneutic procedures (Marino

2006, 2010). It is also less concerned with drawing a new branch into the *critique génétique* in order to extend philological processes concerned with critical editions of books to software.[6] Rather, it is a matter of (further) developing a methodology on a pragmatic level that would allow a software code to not only be executed and applied, but also to be subjected to critical readings. The aim would be to make the algorithms themselves readable through extensive comments, reflections, references and, if necessary, modifications. This methodology does not only serve a didactic purpose, insofar as an understanding of programme structures could strengthen digital literacy, which is sorely needed in the humanities and jurisprudence. As in the transmission of elaborate reading skills using classical academic methods, such as discourse analysis, deconstruction or hermeneutics, the overarching goal is to foster a critical faculty that would be able to understand, disclose, classify, contextualize and explain individual programming steps in order to counter the much invoked power of algorithms. The critique would aim at the shared software code itself. Digitally literate users – not unlike legal scholars in Justinian times – could insert their understanding of the code by commenting on it, explaining it to other readers, discussing certain problems, or hinting at crucial steps in the operational chain.

As distinct from *critical code studies*, this proposal of a source code critique does not call for checking programme structures and command lines for eventual metaphors, figurative speech or ambivalent meanings. Rather it suggests an extensive interplay of code and commentary in order to make the algorithms themselves more transparent and thus more comprehensible, i.e. to prepare them by means of explanatory comments in such a way that interventions and modifications in the program flow can be facilitated (cf. the commentary in the main window of fig. 0). The critical acts in the method of source code critique consist of an elaborate reading of the source code as a first step, and in the second step of commenting on the operational chain, and, finally, if needed, of modifying the algorithms by implementing alternatives to the existing code and providing new versions of the software package.

One could object that these skills are mostly only found among software developers and computer scientists. I would argue that it is necessary to acquire a basic understanding of code. As 'digital natives' who have been taught an appropriate level of digital literacy at school start to arrive in the halls of higher education, the cultural technique of coding presents us with the similar need of training as the cultural techniques of reading and writing on an academically advanced level. You do not have to be a literature scholar in order to read a novel. You do not have to be a computer scientist in order to understand and critically read software code. You only have to learn and acquire basic digital skills at college, or autodidactically by studying online tutorials, if you have not learned it already at school. In order to counter the opacity of code, source code critique as a methodological approach might be added to the curriculum, together with the elaborate reading and writing skills that are natural parts of today's academic education. Gradually, mere users of software may turn into literate readers of software with the ability to understand and modify the code; most inevitably they will also become programmers.

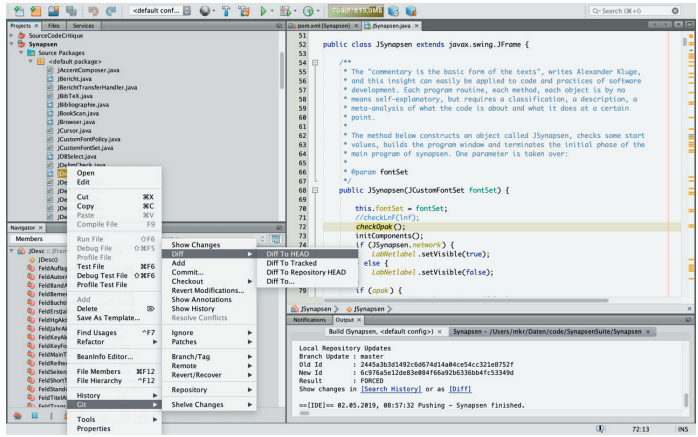

Figure 1: In an *Integrated Development Environment* (IDE) code, commentaries, version control and many other functions are combined in order to help facilitate comprehensive coding

The idea of software development as a kind of philological critique is by no means new. It has been developed by none other than the author of the epochal programming bible, *The Art of Computer Programming* (1968-2025), by Donald E. Knuth. With the ambiguous title 'Literate Programming', Knuth (1984) proposed to write the source code in such a way that it would not only include the commands in the respective programming language, but also the individual instructions and program structures. These would be intensively described and commented on by the developer at the same time. The source code would thus contain the individual commands and data structures together with their documentation or philological apparatus. In this way, algorithms could become transparent and easier to understand, not only for their authors, but also for later readers and editors. A code in the sense of literate programming would be transformed into a text that would be readable for comprehension. The effects of such literate programming could be remarkable: the close reading of an algorithm in a higher programming language becomes an almost textual-philological procedure, which could expose at any moment the state of known knowledge, its references, its hidden structures, its steps of abstraction, and the flow of data in time. They would make the algorithm more transparent and comprehensible. The debugging of an algorithm in this sense would then be nothing more than a deep hermeneutical reading of what is written in the code in order to find possible errors, the correction of which would make the text 'run' again.

Law, literature, code and their critiques are, of course, structurally more closely connected than a first glance would imply: being gripped by a literary text can either be followed by indulgence, or literary insights can be used to analyse how this feeling of being gripped was an effect of authorial design. Understanding coding as a cultural technique makes it possible to critically counter the subjection to algorithmic structures and software-implemented affectations and affectedness. It opens up a view of algorithms in their constructedness. In order to recognise how exactly a user is used by the software, it remains necessary to be able to decipher, comprehend and critically uncover the design and construction of an algorithm.

How can such a critical reading be achieved? This is also a question about a medium; in this case, it refers to the so-called Integrated Development Environment (IDE), which serves as the central medium of criticism, i.e. a programming environment that combines all important tools for analysis, research, error tracing, code production and distribution (see fig. 1). But an IDE is not just a place where mistakes are corrected. It is also the place of discourse, editing, and proofreading of the code (not necessarily by another person), insofar as comments are inserted here, which are then further processed to document the algorithms. It is not only the place where the coherence, effectiveness or elegance of the code is checked and improved. IDE is also the place where one's own modifications to the software are recorded using version control commands, or from where other modified parts of the programme are inserted into one's own local files. These processes of file exchange are easily made possible by the fact that each IDE (like *Eclipse* or *NetBeans* or *IntelliJ IDEA*) is linked to (de-)centralised software repositories like github.com, where developers can obtain other (open) source software packages and make their own code modifications available to these repositories. The developer becomes both a globally connected reader and a writer of software code at the same time. An IDE therefore serves as a decentralized distribution platform where the user's position constantly oscillates between being a creative developer and a critical reader. Whereas the programmer/critic first retraces the code in its capillary ramifications in order to relate it to the overall programme (a hermeneutic operation), in the next moment he/she becomes the annotating and commenting reader in order not to lose the insights gained. The reader simultaneously becomes an intervening writer, true to the old legal-historical maxim that commentary is the basic form of both text and code (Vismann & Krajewski 2007).

An IDE serves as a media milieu for coding, providing an environment in which algorithms thrive, are produced and optimized, and finally made comprehensible, understandable and transparent within the framework of a source code critique. Source code critique serves as the key skill for opening, processing, and deciphering codes

- not only in the context of the Finnish court case, but for almost all purposes of digital criticism.

Endnotes

1. The details of the case can be accessed here: <www.yvtltk.fi/material/attachments/ytaltk/tapausselosteet/45LI2c6dD/YVTltk-tapausseloste-_21.3.2018-luotto-moniperusteinen_syrjinta-S-en_2.pdf>.
2. For a cultural historical genealogy of the command as an act and process, see Canetti (1960/2000). The Nobel Laureate discusses the impulse of fleeing as the primordial instance of the command.
3. Al-Hwarizmi's treatise was later latinized by Leonardo Pisano (Fibonacci) in his *Liber Abaci* 1202.
4. For a derivation and adaptation of the concept of the chain of operations in media theory and cultural techniques research, see the critique of Schüttpelz (2008); Heilmann (2016), as well as the subtle and insightful critique of the critique by Schüttpelz (2017).
5. The argument here refers only to classical algorithmic designs where the programme is strictly deterministic, i.e. it always produces the same output, and the IDE allows the algorithm to be traced at any time whilst the programme runs. In contrast, so called deep -earning algorithms are - by design - mostly not comprehensible to the software developers themselves. Their opacity produces a situation where algorithms intentionally are and act out of control. This aspect of the so-called *algorithmic governance* is not discussed here. For a survey of legal concerns within the realm of algorithmic governance see, e.g. Coglianese and Lehr (2019).
6. See for first approaches Hiller (2014), for philological editions and the *critique génétique* more generally, see Grésillon (1999).

References

Belliger A and D J Krieger eds 2006 *ANThology. Ein einführendes Handbuch zur Akteur-Netzwerk-Theorie* transcript Verlag Bielefeld

Canetti E 1960/2000 *Masse und Macht* Fischer Taschenbuch Verlag Frankfurt am Main

Coglianese C and D Lehr 2019 'Transparency and Algorithmic Governance' *Administrative Law Review* 71: 1-56

Fry H 2018 *Hello world. Being human in the age of algorithms* W.W. Norton & Company New York

Grésillon A 1999 *Literarische Handschriften. Einführung in die critique génétique* Peter Lang Verlag Bern

Heilmann T 2016 'Zur Vorgängigkeit der Operationskette in der Medienwissenschaft und bei Leroi-Gourhan' *Internationales Jahrbuch für Medienphilosophie* 2/1: 7-30

Hiller M 2014 'Diskurs/Signal (II). Prolegomena zu einer Philologie digitaler Quelltexte' *editio. Internationales Jahrbuch für Editionswissenschaft* 28: 192-212

Kneer G, M Schroer and E Schüttpelz eds 2008 *Bruno Latours Kollektive* Suhrkamp Verlag Frankfurt am Main

Knuth D E 1984 'Literate Programming' *The Computer Journal* 27: 97-111

Latour B 1991/2006 'Technologie ist stabilisierte Gesellschaft' in Belliger et al 2006: 369-397

Latour B 2002 *Die Hoffnung der Pandora. Untersuchungen zur Wirklichkeit der Wissenschaft* Suhrkamp Verlag Frankfurt am Main

Leroi-Gourhan A 1965/1980 *Hand und Wort. Die Evolution von Technik, Sprache und Kunst* Suhrkamp Verlag Frankfurt am Main

Marino M C 2006 *Critical Code Studies* <electronicbookreview.com/essay/critical-code-studies/>

Marino M C 2010 *Critical Code Studies and the electronic book review: An Introduction* <http://electronicbookreview.com>

Mayer-Schönberger V and Cukier K 2013 *Big data. A revolution that will transform how we live, work, and think* Houghton Mifflin Harcourt Boston

Saxer D 2014 *Die Schärfung des Quellenblicks. Forschungspraktiken in der Geschichtswissenschaft 1840–1914* De Gruyter Oldenbourg München

Schüttpelz E 2008 'Der Punkt des Archimedes: Einige Schwierigkeiten des Denkes in Operationsketten' in Kneer et al 2008: 234-58

Schüttpelz E 2017 'Die Erfindung der Twelve-Inch der Homo Sapiens und Till Heilmanns Kommentar zur Priorität der Operationskette' *Internationales Jahrbuch für Medienphilosophie* 3/1: 217-34

Vismann C and Krajewski M 2007 'Computer-Juridisms' *Grey Room. Architecture, Art, Media, Politics* 8/29: 90-109