JOURNAL OF
CROSS-DISCIPLINARY RESEARCH
IN COMPUTATIONAL LAW [CRCL]

# Source Code Criticism
# On Programming as a Cultural Technique and its Judicial Linkages

Prof. Dr. Markus Krajewski*

**Abstract**

Despite the growing body of research into cultural techniques, questions regarding the digital, such as the operation of algorithms, have remained underexplored in this field of media studies. I will meet this desideratum by examining programming as a cultural technique. Similar to methods in Critical Code Studies (such as those by Mark Marino) this intends to situate algorithms in a discursive and historical context by elucidating code through systematic commentary, and making software transparent by critical analysis. The following is an attempt to shine some light into the black box, first by examining the varying modes that create inaccessibility, making it difficult to understand and reconstruct code, and secondly, by proposing a method that could remediate the opacity of algorithms. My approach aims to narrativise, historicise, and discursivise code by means of extensive commentary, in order to provide a lever that opens the

**Keywords:** Source Code Criticism, Critical Code Studies, Cultural Techniques, Programming as a Cultural Technique, Commentary, Commentaries as a Judicial Practice, Commentaries as a Philological Practice, Commentaries as a Coding Practice, Hermeneutics of Code

**Replier:** Katja de Vries • Senior Lecturer/Associate Professor, Uppsala University, Sweden. katja.devries@jur.uu.se.

*Journal of Cross-disciplinary Research in Computational Law*

DOI: pending

www.journalcrcl.org

* Professor of History and Theory of Media, University of Basel, markus.krajewski@unibas.ch

## Obscure acts

It is possible to sue your neighbour if you are unable to come to terms, and you can even bring a whale to trial.[1] But it is much more complicated to prosecute algorithms. A great deal of effort is required to bring a piece of code to trial. First, its authorship is not necessarily linked to a legal person. It is also difficult to clarify the facts and understand how the software was built and how it runs – never mind the challenge of changing the algorithm itself.[2] There is a ubiquitous but nevertheless obscure metaphor used to denote the difficulty of understanding an algorithm's function, steps and our ability to grasp its effects. To express our helplessness in the face of the oversized influence of algorithms, people usually speak of a "black box" within which the software unfolds its functions.[3] Something, *data X*, for example the words "I am", falls into an opaque receptacle where it is processed without outside observation, so that almost nobody knows how in the end *data Y*, for instance the finished sentence "not Stiller", has found the light of day.

The following is an attempt to shine some light into the black box, first by examining the varying modes of creating inaccessibility that make it difficult to understand and reconstruct code. This paper/article then proposes a method that could act as a remedy for the opacity of algorithms. First, however, we shall look at the historical development and linguistic effects of the metaphor of the black box, in particular in comparison to the law, which some have suggested is structurally similar to the functioning of software code.[4]

In his history of the black box, Philipp von Hilgers reconstructs two complementary foundational moments for the term; one that is material and connected to a physical object and one that is conceptual and was used in the budding field of cybernetics. The idea of the black box as a technical device whose functioning we do not understand can be dated very precisely to an instance of technological transfer. In September 1940, the British chemist and academic executive Henry Tizard sent the blueprints and schematics of a new kind of radar equipment developed in Great Britain, along with a prototype of this "magnetron", to a research and development team at MIT in Cambridge, Massachusetts. The black anodized magnetron was sent in an equally black metal deed box that became eponymous for a technical device whose input and output is known, but whose technical signal processing remains obscure and opaque.[5] The metaphor of the black box began with such a box (the magnetron) transported within another black box, whose function in this first transmission (from England to the USA) was quite clear, even if the object itself was encapsulated. "The black box is not a package that one need only open to see its content. It contains more black boxes."[6]

Just a few years later, the term "black box" was established as a concept in the newly forming field of cybernetics. The term took root at an informal meeting at the Institute for Advanced Study in Princeton, New Jersey on January 6-7,

[1] See David Graham Burnett, *Trying Leviathan. The nineteenth-century New York court case that put the whale on trial and challenged the order of nature* (Princeton University Press 2007).

[2] One case in which legal recourse was possible occurred in Finland, where a man from a rural area was denied a microcredit by an algorithm on the basis of that characteristic, see AlgorithmWatch, *Automating Society Report 2019* (2019) and Markus Krajewski, 'Hilfe für die digitale Hilfswissenschaft. Von den Digital Humanities verspricht man sich wahre Wunder, obwohl sie nur eine einfache Hilfswissenschaft sind' [2019] (85) Frankfurter Allgemeine Zeitung N4, 119-121.

[3] See, as an example that is almost a diagnosis of our times, Frank Pasquale, *The Black box society. The secret algorithms that control money and information* (Harvard University Press 2016).

[4] See Lawrence Lessig, *Code. Version 2.0* (2. Auflage, Basic Books 2006); and for a counterargument, with a nuanced look at legal elements of computerized systems Cornelia Vismann and Markus Krajewski, 'Computer-Juridisms' (2007) 8(29) Grey Room Architecture, Art, Media, Politics 90; for the description of certain legal processes as black boxes see, e.g., Megan Wright, Shima Baradaran Baughman, and Christopher Robertson, 'Inside the Black Box of Prosecutor Discretion' (2022) 55 UC Davis Law Review 2133; Lauren Sudeall and Daniel Pasciuti, 'Praxis and Paradox: Inside the Black Box of Eviction Court' (2021) 74(5) Vanderbilt Law Review 1365; Wim De Mulder and others, 'Are Judges More Transparent Than Black Boxes? A Scheme to Improve Judicial Decision-Making by Establishing a Relationship with Mathematical Function Maximization' (2021) 84(3) Law and Contemporary Problems 47.

[5] Philpp von Hilgers, 'Ursprünge der Black Box' in Ana Ofak and Philipp von Hilgers (eds), *Rekursionen. Von Faltungen des Wissens* (Wilhelm Fink Verlag 2010) 143–145.

[6] Kathrin Passig, 'Fünfzig Jahre Black Box' (2017) 71(823) Merkur Deutsche Zeitschrift für europäisches Denken 16, 23.

1945, where participants, including Norbert Wiener and John von Neumann, made use of the metaphor. Warren McCulloch, one of the intellectual fathers of artificial intelligence, remembers a passionate duel between von Neumann and Wiener, which he dated incorrectly as occurring in the winter of 1943-4.[7] The two debated about ways to discover the function of a black box — in this hypothetical case, war machinery captured from the Germans — without opening it. After all, it is paradoxical. Once a black box is opened and light falls into it,the contents then become transparent and the 'black' disappears. One somewhat delayed result of the arguments at Princeton, alongside the series of Macy Conferences that began the following year, was the 1956 Dartmouth workshop that sparked the development of artificial neural networks (ANN). We return to ANN an architecture of knowledge, below. In the meantime, it is important to note that the metaphor of the black box had its start in a debate about the opacity of German war-booty – an obscure box whose function cannot be determined directly, but only through the detour of the experimental input and output of varying electronic currents or data streams.

Von Hilgers also points out that the concept of the black box has advantages for computer scientists and software developers, especially with a concept known in object-oriented languages as "encapsulation".[8] As long as the interface is clearly defined and input and output are specified, a great deal of time can be saved by not having to examine the encapsulated methods and algorithmic structures. This means that hidden elements within programs and software libraries, when they have a clearly defined use and explicit application programming interfaces (API), can remain closed regarding their architecture and may have obscure internal functions. As encapsulation in software libraries, repositories, and packages is understood as stabilization, it may not necessary to reveal their construction. Accordingly, in software development, the Schwarzgerät or what Thomas Pynchon described as the *terminus technicus* for the black box in *Gravity's Rainbow*[9] – is meant less to obfuscate and more to simplify by reducing complexity.

The analogy with written laws, whose application and interpretation is often more interesting than the details of their construction, genesis, and function, is easily apparent. Although their original wording is easily accessible, it can still be claimed that the text of laws is covered by a veil of opacity. Their foundation is encapsulated: the constitution is not explicitly mentioned in the wording of individual laws, just as pre-existing methods, or the intended functions, are encapsulated in software libraries, and equally as impossible to reach. And despite their publication, laws also exclude something else inscribed in their functioning, for example the issues that played a role in their writing. Which lobby interests played a part in formulating the text? What are its implicit assumptions? How do the technicalities function and do they prove themselves when used? Although their text is open to the public in full, in many ways — and not only for laypeople — laws operate in an opaque manner.[10]

## Seeing through the Schwarzgerät

The power of algorithms results from a variety of opacities that manifest themselves in different forms. In a first step that makes absolutely no claim to be a systematic survey of the field, a heuristic approach helps us delineate four cases that show why algorithms are so difficult to grasp and how the respective problems can be linked to a legislative context.[11]

[7] See Warren McCulloch, 'Recollections of the Many Sources of Cybernetics' (1974) 6(2) ASC Forum and, for the correction of the date Hilgers (n 5) 150, note 42.

[8] See, e.g., John C Mitchell, *Concepts in programming languages* (Cambridge University Press 2004) 418, 242; see also Erich Gamma and others (eds), *Design patterns. Elements of reusable object-oriented software* (Addison-Wesley professional computing series, Addison-Wesley 1995) 42 ff.

[9] In this case, the load encapsulated in the mysterious weapon turns out to be human: Gottfried shrouded in Imipolex G, see Thomas Pynchon, *Gravity's Rainbow* (Bantam Books 1974).

[10] As it is, e.g., to a certain degree in the decision making of eviction courts in Georgia, as analysed by Sudeall and Pasciuti (n 4).

[11] On other cases meant to prevent penetration of the black box and the corresponding reasons for the unintelligibility of software or the ubiquitous slowness of thought, problems with synchronization and the complexity and size of code, etc. see Passig (n 6).

### *Translation*

As a problem-solving method, algorithms are in fact based on a triad of functionality, understandability, and elegance.[12] But these fundamental principles are not necessarily mirrored in every fragment of code. It is difficult to live up to this ideal under the pressure to find a solution and with the limited resources that dominate in day-to-day working life. More importantly, in most cases algorithms resist readability and hence replicability, because not all software makes its source code, or the sequence of individual commands written in "plain text", *[Klartext]*[13] available to a higher programming language. Yet only when the source code is available can, depending on the language used, the logic of the program be decoded, with more or less effort depending on how abstract the commands, data structure, and routines are: Are they written in English or in Mandarin? Are the descriptions consistent, common, and meant to be understood? Furthermore, one and the same program code can be almost impossible to decipher when it does not use generally understandable terms for its commands (see Listings 1 and 2), but its algorithm is immediately easier to comprehend as soon as such terms are inserted (Listing 2).

Not until a second step, compiling, where the translation of the source code into the binary code that the machine can execute, is the algorithm further obfuscated or locked into a black box that cannot easily be opened in retrospect. By translating the algorithm into executable code (turning it into a *.bin or *.exe file), it is encapsulated and locked into its box where it can no longer be classified.[14]

There is a legal equivalent to closing code in this way, one that goes back to juridical practice in the Roman Empire in the sixth century CE: the pandects, to this day one of the fundaments of civil law. This compendium or digest, as it is also called, of varying legal cases and commentary acted as a memory of judgements and decisions and at the same time as a set of rules. At the behest of the emperor Justinian in 533 CE, these materials were brought together in a book. This codex bundled the opinions of chosen Roman legal scholars of the Republic and the Empire and codified them into law. In the process, most lengthy commentaries were left out when the book's cover was shut — the literal meaning of codifying — and its contents elevated into law.

```
private long rhksog(int lmvlkmvlsdf) {
  if (lmvlkmvlsdf <= 1)
    return lmvlkmvlsdf;
  return rhksog(lmvlkmvlsdf – 1) +
      rhksog(lmvlkmvlsdf – 2);
}

private long asdfasdfsadffasfsad(int herttjrtzhtr) {
  long gdgijimkfe = 0;
  for (int dsfidsjfi = 0; dsfidsjfi <= herttjrtzhtr;
      dsfidsjfi++)
    gdgijimkfe = rzsretsert(herttjrtzhtr);
  return gdgijimkfe;
}
```

**Listing 1:** *one and the same code in two styles: abstract at the top…*

```
private long fibo(int n) {
  if (n <= 1)
    return n;
  return fibo(n – 1) + fibo(n – 2);
}

private long fiboWithoutRecursion(int number) {
  long j = 0;
  for (int i = 0; i <= number; i++)
    j = addFibonacciS(i);
  return j;
}
```

**Listing 2:** *… and for hermeneuts below.*

[12] Donald E Knuth, 'Computer Programming as an Art' (1974) 17(12) Communications of the ACM 667, 670 informed by Jeremy Bentham's utilitarian concept of taste and style, goes even further in his software development and aims for beauty. Bentham, by the way, coined the term codification for the legal practice in an anonymous pamphlet, published in 1776, see Charles Noble Gregory, 'Bentham and the Codifiers' (1900) 13(5) Harvard Law Review 344, 344

[13] See Geoffrey Winthrop-Young, *Friedrich Kittler zur Einführung* (Junius Verlag 2005) 59, 62 ff.

[14] Joasia Krysa and Grzesiek Sedek, 'Source Code' in Matthew Fuller (ed), *Software studies. A lexicon* (The MIT Press 2008); Markus Krajewski, 'Against the Power of Algorithms. Closing, Literate Programming, and Source Code Critique' (2019) 23 Law Text Culture 119, 123

The commentaries became (illegitimate) explanations that lacked the force of law.[15]

But such enclosing or codification is by no means irreversible. Just as Roman law was renewed or modified from time to time, there are tools with which translation into unreadable machine code can be at least partially reversed. Even executable code can be restored to its original state, not least because a computer is a deterministic machine. This practice is called "reverse engineering" and restores the example in Listing 1 to the state shown in figure 1.

What figure 1 shows, at a glance, is that the code no longer seems as graspable as before: it is quasi naked. It lacks the commentary and explanatory structures visible on the left (displayed in light grey colour) -- for a good reason. The machine, resp. the compiler, does not read in the commentary as it contains no commands relevant for executing the code. The compiler, of course, follows a different mode of reading than the user trying to understand the source code. That is, the compiler only interprets the commands – the algorithmic structures of the code – whereas the human user reads at least on two levels, the algorithmic structure as well as the commands intended to elucidate, expand and explain what is laid down in the commands. All that has been reconstructed by the practice of reverse engineering are these functional elements, methods, and data structures, though with their original denomination; the method fiboWithoutRecursion(n) is still meaningful, instead of being renamed as opgdnwdukbsdkfbue(n) which would imply a further escalation of obfuscation. The substitution of meaningful terms with nonsense strings in legal contexts would, probably, not render any helpful juridical practice.

What could be the legal equivalent of reverse engineering? An amendment? More likely an appeal, which, like reverse engineering, requires considerable additional effort. A decision is disputed and a new court must try the case again from a new perspective. The law must be interpreted one more time in order to reach the same or a different verdict. That in turn means again looking one by one at each procedural step, micro-decision, partial argument, method, and interpretation, all of which must be reread and reconstructed to adjust the argumentation and conclusions.

Though analogies are the fundamental method when generating new decisions in common law, the analogy between black boxes and obfuscation in algorithms and in law, naturally, is limited. One major objection might be, that in order to execute an even very obfuscated and encapsulated–code snippet – every single step must be made explicit to the compiler. Nothing is hidden or obscured in the most far-fetched modules of software libraries, nothing will be kept encapsulated or in compressed software libraries during compilation, since the compiler needs to know what to do by pursuing all the commands and its references line by line. If encapsulation in software development should resemble the gating and cutting off of concrete law cases during the process of abstraction in the civil law, here, the black box metaphor also fails, because for ruling according to an established abstracted law one does not need to know all the cases that have led to the formulation of this certain statute. The compiler, however, must be fed with *all* the commands hidden in encapsulated modules, the most abstract as well as the most concrete. In the end, both systems, code compilation as well as jurisdiction operate on a transparent basis where *all* the codes must be — at least theoretically — available and made explicit, i.e. all the information is available to the highest instance.[16] The 'normal' user, however — whoever that is in the interaction with the computer/the law — can be excluded from this transparency by strategies of obfuscation, encapsulation, and codification.

### *Streams*

Another factor that makes algorithms inaccessible is more recent: the infrastructure of digital worlds. Data streams are processed chiefly on the internet. Data, including executable codes, are no longer require local storage. They are relocated to the cloud or to another external location from where they can be fetched as needed, that is to say 'made available' in the form of a local data stream. This is an ephemeral process — in keeping with the nebulous cloud metaphor. There is no explicit, — or for the user,

[15] See Markus Krajewski and Cornelia Vismann, 'Kommentar, Code und Kodifikation' (2009) Frühjahr 2009 Zeitschrift für Ideengeschichte 5, 7 ff.
[16] For the compiler as sovereign, i.e. the most powerful instance in coding, see Vismann and Krajewski (n 4) 97 f.

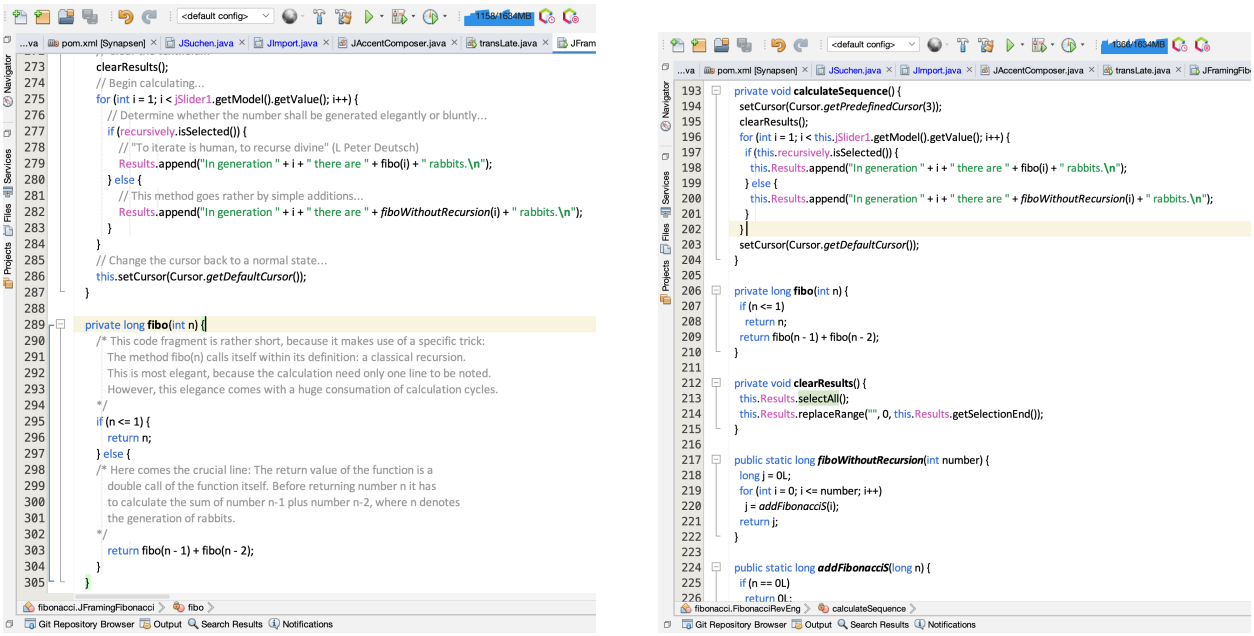

**Figure 1:** *The same program, left in the original, right reverse engineered*

transparent — local storage location planned for such data streams. Whatever has no location remains inaccessible, unaddressable, and hence obscure. It is impossible to get hold of data without an address. The algorithms are encapsulated again, this time in data streams that are not fully available, but work locally in real time and then disappear again. Platforms like Spotify or Netflix act due to this logic; they provide their data as streams not to be stored and accessed or even owned (in the legal sense) on the user's local file system. Instead, they are handled as ephemeral entities. Once watched or listened to they disappear, without being further analysed or scrutinized.

Such difficulties can already be found in presocratic fragments: We cannot step twice into the same river.[17] When dealing with data streams, the fundamental problem again remains their inception. Where can we begin to understand a data stream, when we do not know what was at the onset? Without a beginning, no code can start or be executed, never mind restored to its original state. With streaming, both the run time and the starting time of the code's execution become crucial.

### *Tiers*

A completely different type of inaccessibility as regards algorithms can be found in an issue that is currently at the fore of debates in and on computer science and its sociotechnical impact: artificial intelligences and their social, habitual, ethical, economical, and last but not least legal consequences. The primary difficulty here — in contrast to the case of conventional software architectures and systems — is that not even the best computer scientist can explain exactly how decisions are made within artificial neural networks (ANN), much less reproduce the results on a micro or macro level. This is due to the architecture of ANN, which have a memory system of artificial neurons as an electronic simulation of the human brain. This memory must be trained with data — whether images, texts, or voices — until enough knowledge has been gathered of the, so to speak, genus and species of the input data. On a purely technical level, this knowledge is saved in vector spaces as probabilities of individual nodes that react with other nodes at an input signal. When the training is finished, the knowledge is frozen: the so-called model can then only react to user queries by reproducing the information fed to its 'brain,' but not by incorporating new

[17] Jaap Mansfeld (ed), *Die Vorsokratiker. Griechisch / Deutsch* (Erw. Neuausg., Nachdruck Auflage, Philipp Reclam jun 2012). Fragment 91, see also 12, 49a.

information during the interaction. However, the model does not always react the same way to specific input, but differently and to a certain degree unpredictably, because the decision path through the layers of artificial neurons is probabilistic and does not operate with fixed trajectories or certainties in an if-then structure. The system's architecture follows a "connectionist paradigm"[18] in which individual artificial neurons can come together in a variety of different constellations. That means that the same input into an ANN or a black box will prompt different results, even in processes that follow in quick succession.

The best legal analogy for an ANN might be an enormous, many-tiered, closed digest. This digest contains myriad cases, judgments, and legal trials, but these are not verbatim nor systematically ordered with a registry and index. Instead, it is jumbled and fragmented, every fragment is given a different relevance regarding how it connects to other, neighbouring, particles and again to their neighbours. If we take this analogy further, such a book of law proves to be less than useful, for every (verbatim) input delivers a different output as a result of the micro-decisions that depend statistically on one another and that take place at every node of the hidden network layer, together leading to ever-different verdicts.

The architecture of an ANN, which enables so-called "deep learning" (a recursive process in which neural networks are deployed that are linked to themselves and to lower and higher tiers or, simplified, architectures with built-in error correction),[19] is made up of more than just its operation within a black box. A typical ANN — in the tradition of its basic building block, the perceptron — consists of three tiers: input, a hidden layer, and output. And this hidden layer, which is made up out of many interconnected layers of artificial neurons,[20] represents as it were — as in its primal scene, the transport from Great Britain to Massachusetts of a black magnetron in a black box — the black box in the black box.

The knowledge of an ANN is therefore not only encapsulated in many black boxes, it is also atomized or dispersed numerically in vector spaces, distributed among many tiny electronic memory elements (artificial neurons), linked by nothing other than a statistical value as a the probability of transition to their respective neighbours and neighbours of neighbours. At the centre of the encapsulated black box of artificial neural networks is fog, a particularized form of random micro-decisions that deliver remarkable achievements despite their fragmentation.[21]

The problem of decision making in black boxed AI systems has long been a topical issue, ever since the first trials of autonomous cars ran into legal difficulties that remain unresolved to this day: Who decides how to react in a harmful traffic situation when all options are bad, the car, the driver, the algorithm, the manufacturer, the software developer? And who bears the responsibility for the decision? This problem is exacerbated by the aforementioned unpredictability with which an ANN makes its decisions. This can only be measured by certain test procedures in which a whole series of similar decisions are also documented. However, this contingency of ANNs can also be helpful, for instance when it comes to the problem of legal uncertainty. If judges, like ANNs, are viewed as black boxes, the path of their decision making could be made transparent in a manner similar to the case of ANNs, in that judges make their alternatively considered options explicit as well.[22] The — to a certain degree — unpredictable functioning of a black box would thus be transformed from problem to virtue.

Behind this, however, lies an even more fundamental problem of understanding: If not even computer scientists can reconstruct the functioning of their own systems, which

[18] Hannes Bajohr, 'Algorithmic Empathy: Toward a Critique of Aesthetic AI' (2022) 30(2) Configurations 203, 219 ff.

[19] See Ethem Alpaydın, *Machine learning. The new AI* (MIT Press essential knowledge, The MIT Press 2016) 85 ff.

[20] Pedro Domingos, *The master algorithm. How the quest for the ultimate learning machine will remake our world* (Basic Books 2015) 101.

[21] For a visual example see Emily Lanza, *Who Painted Rembrandt? Copyright and Authorship of Two Rembrandt Portraits* (Published: www.thelegalpalette.com, The Legal Palette, 2018); for textual examples see the results of GPT-3, designed by Tom B Brown and others, *Language Models are Few-Shot Learners* (Published: arxiv.org, arXivorg, May 2020); e.g. with results like this: https://www.gwern.net/GPT-3, recently optimized for dialogue with ChatGPT.

[22] See De Mulder and others (n 4) 48,63, who also discuss the more general aspects of black boxes in jurisdiction and its similarities to artificial neural networks.

still operate within deterministic machines, how might a detailed critical understanding of this technology be possible? The problem, it must be said, is also recognized within the field of computer science and is currently receiving much attention under the denotation "explainable AI" (XAI).[23] One central finding is that while a black box is operating, it can in principle no longer be opened — like Schrödinger's cat in the box. Transparency can be created solely on the conceptual level, which leads us back to source code. Only on this concrete level, that is before the actual execution of the code, can decision-making paths be understood or reconstructed. But a certain amount of preparation is needed tto be able to work on source code. The path to understanding — in classical tradition — follows the mileposts of a philological and historiographical virtue: reading the sources. What does this mean concretely?

## The remedy: source code criticism

Even code that is freely available is not necessarily easily readable. Source code that is not behind barriers or hidden by other obfuscation or nebulization still resists readers other than geeks, hackers, and nerds. Yet it is more urgent than ever that genuine computer literacy exist outside these groups. In the humanities and in law, scholars must master and pass on the skills necessary to research not only complex philosophical, legal, and literary texts, but also code — from a critical perspective.

This is the starting point of programming as a cultural technique — my proposal for a new methodology for dealing with code beyond the computer sciences. It consists of a process I here call "source code criticism". The code of a software project should be brought together with extensive explanatory commentary to form a work that is a combination of text and code and that, in the best-case scenario, can present its results in the form of both a book on coding with its cultural explanation and as a software application.

Cultural techniques, with their interaction of targeted physical gestures and the use of objects such as tools, instruments, or other media, act in a manner that has a specific cultural impact. Research on cultural techniques therefore investigates the practices of those processes that are constitutive for culture in relation to their mediality, including the procedures, gestures, and tools involved in their historical development as well as their cultural and epistemic foundations.[24] Cultural techniques always have an additional aesthetic component that goes beyond their functionality alone, something that Donald Knuth for example demands of programming — to take account of questions of style, elegance, and not least beauty when developing code.

Despite increased interest in and the growing institutional importance of research into cultural techniques,[25] to date the issue of digital practices such as algorithm development and how it functions as a cultural technique has been mostly ignored. This desideratum must be met by examining coding as a cultural technique. This includes not only the ability to read and write code or develop it for daily software needs, but also to subject code to a critical analysis in its discursive and historical contexts. In the future, this skill will play a key role not only for scholars in the humanities, but also for legal scholars, because these competencies will be needed in more people than just professional software developers, computer engineers, interface designers, and, not least, the self-learning machines themselves. The actions and design of said machines and the invention and training of algorithms must be accompanied by critical reflection and a broader understanding. Only then will it be possible to demystify the spectre of the power of the algorithms, to delimit and to analyze them.

[23] See, e.g., Wojciech Samek and others (eds), *Explainable AI: Interpreting, Explaining and Visualizing Deep Learning* (Lecture Notes in Artificial Intelligence, vol 11700, Springer International Publishing 2019).

[24] See Erhard Schüttpelz, 'Die medienanthropologische Kehre der Kulturtechniken' (2006) 6 Archiv für Mediengeschichte 87; Bernhard Siegert, 'Cultural Techniques: Or the End of the Intellectual Postwar Era in German Media Theory' (2013) 30(6) Theory, Culture & Society 48; Bernhard Siegert, *Cultural Techniques: Grids, Filters, Doors, and other Articulations of the Real* (Geoffrey Winthrop-Young ed, Fordham University Press 2015).

[25] See, e.g., Geoffrey Winthrop-Young, 'The Kultur of Cultural Techniques. Conceptual Inertia and the Parasitic Materialities of Ontologization' (2014) 10(3) Cultural Politics 376; Geoffrey Winthrop-Young, 'Discourse, Media, Cultural Techniques: The Complexity of Kittler' (2015) 130(3) Moden Language Notes 447; Geoffrey Winthrop-Young, 'Siren Recursions' in Stephen Sale and Laura Salisbury (eds), *Kittler Now: Current Perspectives in Kittler Studies* (Polity Press 2015); Geoffrey Winthrop-Young, 'The Kittler Effect' (2017) 44(132) New German Critique 205.

Codes can only be domesticated through comments.[26] Concretely, for scholars of text this means learning to take a critical distance to algorithms. It means learning to decode and segment them in order to investigate the way in which they function, their impact and their linguistic particularities, their design and style in order to classify them according to historical, legal, cultural, or political standards. This approach is slightly different from that taken in the field of software studies as propagated by Matthew Kirschenbaum,[27] Stephen Ramsay,[28] Lev Manovich,[29] or Nick Montfort.[30]. The aim is also and importantly to strengthen the digital literacy of scholars in the humanities.[31] Mark Marino in particular, in developing the innovative field of Critical Code Studies, has shown how a discursive reading of algorithms can function, for example in his close readings of the collaborative story *exquisite_code*,[32] and, more recently, of Friedrich Kittler's code production.[33]

By now it should be clear that instead of quantitative statistical or numerical analyses that look, for example, for word frequency (a method used all too often in the digital humanities),[34] programming as a cultural technique instead aims to transfer a core competency of the humanities, critical reading, to the digital world. This means being able not only to understand, classify, and modify algorithms, but also to be able to grasp and comment on them using the termini, theories, concepts, and dispositifs of literary criticism. This form of contextualization is what makes it possible to deconstruct code as if it were a literary text. In marked contrast to this novel approach, the opposite idea — designing software as if it were literature — is by no means new. Donald E. Knuth, author of the epochal book *The Art of Computer Programming* (1968-2025) and of the TeX editing system, had in 1984 already published an article with the equivocal title *Literate Programming* that proposed writing source code from the outset so that it contained more than the commands of its respective programming language.[35] As an alternative, he called for a meta-level in which developers included in-depth description of and commentary on the individual commands. The source code would then comprise not only the individual commands and data structure, but also document the same. In this way, algorithms would be and remain transparent—not only for their authors, but also for readers and developers who came after (see listing 3 on p. 10 and listing 4 on p. 12). It is hardly worth mentioning that this paradigm is not applied in the daily routine of professional software developers.

In the meantime, however, a comparable practice has been established elsewhere that effectively interweaves executable code and commentary, namely in the form of the so-called Jupyter Notebooks, an open-source platform whose format has been developed for the interactive and collaborative analysis of measurement data. Crucially, measurement data and their processing are framed by explanations, metadata, and contextualizations that contribute as so-called Jupyter Narratives to the understanding of the respective analysis steps.[36] The code cannot do without supplementary explanatory material if it is to remain collaboratively comprehensible. A very similar form

[26] Krajewski, 'Against the Power of Algorithms. Closing, Literate Programming, and Source Code Critique' (n 14).

[27] Matthew Kirschenbaum, *Hello Worlds. Why humanities students should learn to program* (The Chronicle of Higher Education, 2009); Matthew Kirschenbaum, 'What Is Digital Humanities and What's It Doing in English Departments?' (2010) 47(150) ADE Bulletin 55.

[28] Stephen Ramsay, *Reading machines. Toward an algorithmic criticism* (Topics in the digital humanities, University of Illinois Press 2011).

[29] Lev Manovich, *Software takes command* (International texts in critical media aesthetics, vol 5, Bloomsbury 2013).

[30] Nick Montfort, *Exploratory programming for the arts and humanities* (The MIT Press 2016).

[31] See also Geoff Cox and Christopher Alex McLean, *Speaking code. Coding as aesthetic and political expression* (Software studies, The MIT Press 2013); David M Berry and Anders Fagerjord, *Digital Humanities. Knowledge and Critique in a Digital Age* (Polity Press 2017); Daniel Punday, *Computing as Writing* (University of Minnesota Press 2015); Mark C Marino, *Critical Code Studies* (Published: electronicbookreview.com, electronic book review, April 2006); Mark C Marino, *Critical Code Studies and the electronic book review: An Introduction* (Published: electronicbookreview.com, electronic book review, 2010).

[32] Mark C Marino, 'Reading exquisite_code' in NKatherine Hayles and Jessica Pressman (eds), *Comparative textual media. Transforming the humanities in the postprint era* (Electronic mediations, University of Minnesota Press 2013) vol 42.

[33] Mark C Marino, *Critical code studies* (Software studies, The MIT Press 2020) 161-197.

[34] As one example: Franco Moretti, *Distant reading* (Verso 2013).

[35] Donald E Knuth, 'Literate Programming' (1984) 27 The Computer Journal 97.

[36] Brian Granger and Fernando Pérez, *Jupyter: Thinking and Storytelling with Code and Data* (Published: /www.authorea.com, Authorea, 2021) 4.

```
private void calculateSequence() {
    /* What seems so interesting about calculating this series of numbers?
  There are two reasons, one is the history of this algorithm, the other is the style, how this algorithm can be implemented.

  1. In 1202, Fibonacci published the Liber Abaci, introducing not only the arabic
       numerals, but also algorithms to occidental mathematics. It contains a famous series of numbers, baptised after him. He
       developed his example by a famous scenario:
      "How Many Pairs of Rabbits Are Created by One Pair in One Year. A certain man had one pair of rabbits together in a
       certain enclosed place, and one wishes to know [...]."

  2. The sequence of numbers can be calculated in two different ways, on the one hand most elegantly by a recursive method, starting
  with the last generation and going back to the first, while it calls itself as a method while it calls itself as a method while it
  calls itself as ... until the initial value n=1 is reached. Then all the numbers are available and can be added to the final sum.

    On the other hand, the value can be calculated just by remembering the last and the last but one value. This is rather easy
  concerning calculating power and ressources, however it takes more code to produce the non–recursive algorithm.

  In the following, both ways are implemented, so the user can compare the differences by walking the code line by line... */

  // Change the cursor to a wait state...
    this.setCursor(Cursor.getPredefinedCursor(Cursor.WAIT_CURSOR));
    // Clear the textfield ...
  clearResults();
    // Begin calculating...
    for (int i = 1; i < jSlider1.getModel().getValue(); i++) {
      // Determine whether the number shall be generated elegantly or bluntly...
        if (recursively.isSelected()) {
          // "To iterate is human, to recurse divine" (L Peter Deutsch)
            Results.append("In generation " + i + " there are " + fibo(i) + " rabbits.\n");
      } else {
       // This method goes rather by simple additions...
            Results.append("In generation " + i + " there are " + fiboWithoutRecursion(i) + " rabbits.\n");
      }
  }
    // Change the cursor back to a normal state...
    this.setCursor(Cursor.getDefaultCursor());
}

private long fibo(int n) {
  /* This code fragment is rather short, because it makes use of a specific trick:
       The method fibo(n) calls itself within its definition: a classical recursion. This is most elegant, because the calculation
       need only one line to be noted. However, this elegance comes with a huge consumation of calculation cycles. */
    if (n <= 1) {
      return n;
  } else {
    /* Here comes the crucial line: The return value of the function is a double call of the function itself. Before returning
    number n it has to calculate the sum of number n–1 plus number n–2, where n denotes the generation of rabbits. */
      return fibo(n – 1) + fibo(n – 2);
  }
}
```

**Listing 3:** *A recursion generating Fibonacci numbers and its explanation*

can be found in Australian legal practice where, within the framework of legislative procedures and regulatory instruments, the process of creating the law is also documented with additional explanatory material (elucidating its origins, reasons and forms of the law to be enacted) in order to serve as an aid both for the subsequent application of the laws and their critical scholarly monitoring.

Literature, law, code, and their criticism are structurally closer than it might at first seem. Just as the cultural technique of reading at an academic level allows the reader to lay bare the manner in which the text was constructed: its rhetoric, its production of affect, its style, and its literary figures of thought, coding literacy can empower practitioners because they would no longer need be subject to algorithmic structures and the dependencies created by software, but would be able to open the black box called "code" and reveal the way in which the algorithm was developed.[37] Such readings include questions of the construction and the poetics of algorithms. There are many paths to solving digital problems. Some of them are as monotonous as a twelve-lane highway crossing California's San Francisco Bay, some have the exploratory potential of the country lane in Arno Schmidt's *Bottom's Dream*, and some rare roads are even like the one still to be found in Knut Hamsun's *The Growth of the Soil*, nothing but a path leading into the wilderness to transform it into a civilized future. In short, in the classification and hermeneutic reading of code, it is key to also investigate intentions, formulations, and questions of style. For in order to recognize how users are used by software, it is necessary to decode the construction and design of algorithms and be able to subject them to critical reflection. This process of the critical reflection of algorithms and the corresponding dedication to coding literacy in the humanities at the most basic level of the code is, in a nutshell, what "source code criticism" entails.[38]

This novel method brings together classic scholarship and historical source code analysis, linking a careful examination and preparation of the material,[39] in this case algorithms, with a theoretically grounded reading that includes critical commentary of program structures and aims for practical functionality. Conceptually, this approach includes on the one hand making source code available in open access repositories such as `GitHub` or, more complicated, through processes such as decompiling or reverse engineering. Further, source code criticism entails the critical reading of code, which, in its dynamic transformations, can be treated like historical sources that, not least because of their many versions,[40] require classification and commentary.

## The power of commentary

Commentary is a particularly central medial practice of programming as a cultural technique. From its classic application in theological exegesis and in legal practice since late Antiquity, commentary has been used to hold up, determine, and vindicate text. Whether the law is religious or judicial, commentary keeps it from becoming inert or incomprehensible; it keeps arguments fluid by underlining particular statements and bringing others into the discourse.[41] Commentary has a similar function in philological text analysis, in the creation of critical editions, and in critique génétique.[42] It points out where the text is unclear or ambiguous, where there are variations or deletions in the original, and so makes transparent the genesis and construction of the text. And, finally, the necessity of commentary can be seen in the context of digital

[37] Annette Vee, *Coding literacy. How computer programming is changing writing* (Software studies, The MIT Press 2017).

[38] Krajewski, 'Hilfe für die digitale Hilfswissenschaft. Von den Digital Humanities verspricht man sich wahre Wunder, obwohl sie nur eine einfache Hilfswissenschaft sind' (n 2).

[39] Daniela Saxer, *Die Schärfung des Quellenblicks. Forschungspraktiken in der Geschichtswissenschaft 1840–1914* (De Gruyter Oldenbourg 2014) 376 ff.

[40] See Markus Krajewski, *Versionskontrolle* (Repositories of Markus Krajewski, githubcom/nachsommer 2020); Markus Krajewski, 'branch, diff, merge. Versionskontrolle und Quellcodekritik' in Jörg Paulus, Andrea Hübener, and Fabian Winter (eds), *Duplikat, Abschrift & Kopie. Kulturtechniken der Vervielfältigung* (Böhlau Verlag 2020).

[41] Vismann and Krajewski (n 4) p.102; Krajewski and Vismann (n 15) 5-9.

[42] Gérard Genette, *Paratexte. Das Buch vom Beiwerk des Buches* (Campus Verlag 1992); Almuth Grésillon, *Literarische Handschriften. Einführung in die critique génétique* (Peter Lang Verlag 1999).

Listing 4: A Very Brief Example of *Source Code Criticism*

```
/* How to get surprised by your digital assistant?
Computers don't do well with true randomness (Rubin 2011). Therefore, the machine, for example, can't provide a truly random card from a collection of index cards. Below, we explore four ways in which surprise can be implemented, based on historic examples of designing serendipity...*/

  public int getRandomInt(int max) {
     /* This method selects a pseudo-random card from the card index collection.
     It takes 'int max' as input, indicating the collection's current amount of cards. */
     // The number to be determined will be chosen from a range between max and this minimal value
     int min = 1;
     // JAVA can now create a pseudo-random integer by this new object:
     Random randomGenerator = new Random();
     // With the result of randomGenerator this method then returns an 'accidentally' chosen number between 1 and the max amount of cards:
     return randomGenerator.nextInt(max - min) + min;
  }

  // Inform the user directly about Mallarmé (1992) and his coup de dés:
  System.out.println("The dice roll ended at " + getRandomInt(JSynapsen.recAnz));

/* Implementing serendipity with Aby Warburg
Since serendipity is even more difficult to implement and at the same time still a young field in information science (McCay-Peet and Toms 2017), we create another unfolding opportunity for 'controlled coincidence' by bringing Aby Warburg's 'law of the good neighbour' to bear (see Krajewski 2017, pp. 99-101): Since 1924, library staff Gertrude Bing and Fritz Saxl, in the Kulturwissenschaftlichen Bibliothek Warburg (K.W.B.) in Hamburg and later in London, set aside similar texts and corresponding thoughts for a given book by constantly rearranging the shelves, thus creating new spatial connections for a book, transforming the previously remote into propinquity through establishing short distances. This creates new links between heterogeneous units of text in close proximity, which can be explored by the reader with the confidence that a knowledgeable hand has curated the surrounding texts half by chance, half intentionally. We simulate this knowledgeable hand by computing similarities between texts in three different ways:
1. by the number of matching terms with which the texts are tagged. The highest match leads to the greatest spatial proximity.
2. by matching the full texts. The entire content of a book is examined with all other full texts for word matches and assigned a characteristic value. Again, the texts with the highest similarities move close together.
3. by an analysis of the quotations in the book. The texts that are cited in the selected book and are also present in the local database are included in the neighborhood, the highest citation frequency determines the closest proximity.
The respective neighborhoods of the selected text can then be illustrated in the form of a sociograph in the diagram. */

  public String[] generateGoodNeighboursByHeadwords(string keyword) {
       /* First we retrieve a list of headwords from the database associated with this card, identified by its keyword like "warburg:1924" [...] This method returns a String array with a list of keyword, representing the neighbourhood of the chosen card. */
       System.reader.println("... This code continues and can soon be followed at
       github.com/nachsommer/interlocutor ...");
  }
```

philology, which applies the methods of philology to software.[43]

In this context, where programming has become a new and innovative form of cultural technique, commentary comprises all of these functions. Yet the method goes one decisive step further to allow commentary its entire epistemic range. One of the greatest epistemic attributes of commentary lies in the systematic transition between levels. There is always a slight gap between text and commentary. Crossing this gap causes an automatic shift in perspective, combined with the necessarily distanced view from the level of commentary upon the object of analysis, whether literary text or code. This shift in perspective includes a moment of self-reflection, in which the commentary privileges a critical examination of the practice or process of writing. By systematically moving between levels, commentary creates an epistemic lever that allows a continuously oscillating perspective on what is written and as such represents an underexploited epistemological instrument. For commentary always acts as the unassuming assistant of reflection, inviting and provoking explanation and plausibilization, exegesis, and links to other texts. The concept of source code criticism uses this oscillating perspective, between operative and explanatory segments, to make the code's commentary into the true text.

The fundamental distinction between code and commentary also resonates with the difference between production and execution of rules in the juridical realm which Laurence Diver discusses in his brilliant paper on computational legalism.[44] As one strategy on how to overcome this gulf Diver advocates for analysing code as text, i.e. considering code as an alterable document rather than something unchangeable as it is given in the "ruleishness" of the legalism paradigm wherein subjecting to the code as law seems inevitable.[45]

The method of source code criticism hence implies two things: on the pragmatic level, that software should not only be executed, but also at times subject to a systematic critical reading, making its algorithms understandable and plausible by means of explanations, reflections, references, and if necessary, modifications. The aim of this method is however not just transparency and intelligibility for the sake of didactics alone, but an understanding of program structures to improve coding literacy. Secondly, on the epistemic level, which in turn goes far beyond the effects intended by by Knuth's (1974) principle of *Literate Programming*, the method aims, by means of extensive commentary, to narrativise, historicise, and discursivise code. In a nutshell: the continuous commentary within the code should become the true text, which means nothing other than writing code as history (as indicated in listing 3). In summary, we are talking about developing a model of source criticism for the twenty-first century and aim to achieve nothing less than setting a new standard for writing code in the humanities.

Accordingly, and not unimportantly, this understanding of code possesses emancipatory potential. Source code criticism means coding literacy at a level that enables reflection on power structures in digital societies. Programming as a cultural technique empowers scholars to understand, classify, and consequently also write code, taking back a certain agency in light of the current power of information technologies.

*An earlier version of this text was translated from the German by Laura Radosh.*

# References

AlgorithmWatch, *Automating Society Report 2019* (2019).

Alpaydın E, *Machine learning. The new AI* (MIT Press essential knowledge, The MIT Press 2016).

Bajohr H, 'Algorithmic Empathy: Toward a Critique of Aesthetic AI' (2022) 30(2) Configurations 203.

[43] See, e.g., Moritz Hiller, 'Diskurs/Signal (II). Prolegomena zu einer Philologie digitaler Quelltexte' (2014) 28 editio Internationales Jahrbuch für Editionswissenschaft 192; Thorsten Ries, '"die geräte klüger als ihre besitzer": Philologische Durchblicke hinter die Graphical User Interface. Überlegungen zur digitalen Quellenphilologie, Studie zu Michael Speiers "ausfahrt st. nazaire"' (2010) 24 Editio Internationales Jahrbuch für Editionswissenschaft 149; Montfort (n 30) as well as the the digital archival projects at the Deutsches Literaturarchiv Marbach.

[44] Laurence Diver, *Interpreting the Rule(s) of Code: Performance, Performativity, and Production* (Published: law.mit.edu, MIT Computational Law Report, 2021).

[45] ibid.

Berry DM and Fagerjord A, *Digital Humanities. Knowledge and Critique in a Digital Age* (Polity Press 2017).

Brown TB and others, *Language Models are Few-Shot Learners* (Published: arxiv.org, arXivorg, May 2020).

Burnett DG, *Trying Leviathan. The nineteenth-century New York court case that put the whale on trial and challenged the order of nature* (Princeton University Press 2007).

Cox G and McLean CA, *Speaking code. Coding as aesthetic and political expression* (Software studies, The MIT Press 2013).

De Mulder W and others, 'Are Judges More Transparent Than Black Boxes? A Scheme to Improve Judicial Decision-Making by Establishing a Relationship with Mathematical Function Maximization' (2021) 84(3) Law and Contemporary Problems 47.

Diver L, *Interpreting the Rule(s) of Code: Performance, Performativity, and Production* (Published: law.mit.edu, MIT Computational Law Report, 2021).

Domingos P, *The master algorithm. How the quest for the ultimate learning machine will remake our world* (Basic Books 2015).

E Gamma and others (eds), *Design patterns. Elements of reusable object-oriented software* (Addison-Wesley professional computing series, Addison-Wesley 1995).

Genette G, *Paratexte. Das Buch vom Beiwerk des Buches* (Campus Verlag 1992).

Granger B and Pérez F, *Jupyter: Thinking and Storytelling with Code and Data* (Published: /www.authorea.com, Authorea, 2021).

Gregory CN, 'Bentham and the Codifiers' (1900) 13(5) Harvard Law Review 344.

Grésillon A, *Literarische Handschriften. Einführung in die critique génétique* (Peter Lang Verlag 1999).

Hilgers Pv, 'Ursprünge der Black Box' in A Ofak and P von Hilgers (eds), *Rekursionen. Von Faltungen des Wissens* (Wilhelm Fink Verlag 2010).

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

Kirschenbaum M, *Hello Worlds. Why humanities students should learn to program* (The Chronicle of Higher Education, 2009).

— 'What Is Digital Humanities and What's It Doing in English Departments?' (2010) 47(150) ADE Bulletin 55.

Knuth DE, 'Computer Programming as an Art' (1974) 17(12) Communications of the ACM 667.

— 'Literate Programming' (1984) 27 The Computer Journal 97.

Krajewski M, 'Against the Power of Algorithms. Closing, Literate Programming, and Source Code Critique' (2019) 23 Law Text Culture 119.

— 'Hilfe für die digitale Hilfswissenschaft. Von den Digital Humanities verspricht man sich wahre Wunder, obwohl sie nur eine einfache Hilfswissenschaft sind' [2019] (85) Frankfurter Allgemeine Zeitung N4.

— 'branch, diff, merge. Versionskontrolle und Quellcodekritik', in J Paulus, A Hübener, and F Winter (eds), *Duplikat, Abschrift & Kopie. Kulturtechniken der Vervielfältigung* (Böhlau Verlag 2020).

— *Versionskontrolle* (Repositories of Markus Krajewski, github.com/nachsommer 2020).

Krajewski M and Vismann C, 'Kommentar, Code und Kodifikation' (2009) Frühjahr 2009 Zeitschrift für Ideengeschichte 5.

Krysa J and Sedek G, 'Source Code' in M Fuller (ed), *Software studies. A lexicon* (The MIT Press 2008).

Lanza E, *Who Painted Rembrandt? Copyright and Authorship of Two Rembrandt Portraits* (Published: www.thelegalpalette.com, The Legal Palette, 2018).

Lessig L, *Code. Version 2.0* (2. Auflage, Basic Books 2006).

Manovich L, *Software takes command* (International texts in critical media aesthetics, vol 5, Bloomsbury 2013).

J Mansfeld (ed), *Die Vorsokratiker. Griechisch / Deutsch* (Erw. Neuausg., Nachdruck Auflage, Philipp Reclam jun 2012).

Marino MC, *Critical Code Studies* (Published: electronicbookreview.com, electronic book review, April 2006).

— *Critical Code Studies and the electronic book review: An Introduction* (Published: electronicbookreview.com, electronic book review, 2010).

— 'Reading exquisite_code', in NK Hayles and J Pressman (eds), *Comparative textual media. Transforming the humanities in the postprint era* (Electronic mediations, University of Minnesota Press 2013) vol 42.

— *Critical code studies* (Software studies, The MIT Press 2020).

McCulloch W, 'Recollections of the Many Sources of Cybernetics' (1974) 6(2) ASC Forum.

Mitchell JC, *Concepts in programming languages* (Cambridge University Press 2004).

Montfort N, *Exploratory programming for the arts and humanities* (The MIT Press 2016).

Moretti F, *Distant reading* (Verso 2013).

Pasquale F, *The Black box society. The secret algorithms that control money and information* (Harvard University Press 2016).

Passig K, 'Fünfzig Jahre Black Box' (2017) 71(823) Merkur Deutsche Zeitschrift für europäisches Denken 16.

Pisano L, *Il liber abbaci. Pubbl. da Baldassarre Boncompagni* (Pisano: Scritti, vol 1, Tipogr delle Scienze Matematiche e Fisiche 1857).

Punday D, *Computing as Writing* (University of Minnesota Press 2015).

Pynchon T, *Gravity's Rainbow* (Bantam Books 1974).

Ramsay S, *Reading machines. Toward an algorithmic criticism* (Topics in the digital humanities, University of Illinois Press 2011).

Ries T, '"die geräte klüger als ihre besitzer": Philologische Durchblicke hinter die Graphical User Interface. Überlegungen zur digitalen Quellenphilologie, Studie zu Michael Speiers "ausfahrt st. nazaire"' (2010) 24 Editio Internationales Jahrbuch für Editionswissenschaft 149.

W Samek and others (eds), *Explainable AI: Interpreting, Explaining and Visualizing Deep Learning* (Lecture Notes in Artificial Intelligence, vol 11700, Springer International Publishing 2019).

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

Schüttpelz E, 'Die medienanthropologische Kehre der Kulturtechniken' (2006) 6 Archiv für Mediengeschichte 87.

Siegert B, 'Cultural Techniques: Or the End of the Intellectual Postwar Era in German Media Theory' (2013) 30(6) Theory, Culture & Society 48.

— *Cultural Techniques: Grids, Filters, Doors, and other Articulations of the Real* (Winthrop-Young G ed, Fordham University Press 2015).

Sudeall L and Pasciuti D, 'Praxis and Paradox: Inside the Black Box of Eviction Court' (2021) 74(5) Vanderbilt Law Review 1365.

Vee A, *Coding literacy. How computer programming is changing writing* (Software studies, The MIT Press 2017).

Vismann C and Krajewski M, 'Computer-Juridisms' (2007) 8(29) Grey Room Architecture, Art, Media, Politics 90.

Winthrop-Young G, *Friedrich Kittler zur Einführung* (Junius Verlag 2005).

— 'The Kultur of Cultural Techniques. Conceptual Inertia and the Parasitic Materialities of Ontologization' (2014) 10(3) Cultural Politics 376.

— 'Discourse, Media, Cultural Techniques: The Complexity of Kittler' (2015) 130(3) Moden Language Notes 447.

— 'Siren Recursions', in S Sale and L Salisbury (eds), *Kittler Now: Current Perspectives in Kittler Studies* (Polity Press 2015).

— 'The Kittler Effect' (2017) 44(132) New German Critique 205.

Wright M, Baughman SB, and Robertson C, 'Inside the Black Box of Prosecutor Discretion' (2022) 55 UC Davis Law Review 2133.

# A reply: to Markus Krajewski, "Source Code Criticism: On Programming as a Cultural Technique and its Judicial Linkages"

**Katja de Vries** • Senior Lecturer/Associate Professor, Uppsala University, Sweden.

katja.devries@jur.uu.se

Texts have an origin and a destination. Texts act, do things, affect. When a text cannot be read any longer, the text becomes a dead letter. Sometimes texts do completely other things than their author intended[46]. As Roland Barthes (1967) famously wrote, the author is dead. Sometimes the author is a diffuse origin that has little to do with the 18th century romantic notion of individual creative genius; and the reader is a nonhuman entity. A DNA sequence is a text that, after being translated into messenger RNA, can be read by ribosomes as building instructions. Who is the author of the DNA text? Richard Dawkin's (1986) blind watchmaker?

In his brilliant paper 'Source Code Criticism' Krajewski writes about the role of commentaries in relation to three types of texts: literature, software and law. In literature and law there is an extensive tradition of commentaries that remove the "veil of opacity" (p. 3) and increases readability. Both law and software are texts that will often be difficult to read for a human reader. In the 11th century Roman legal texts that had been 'dead' for many centuries were made readable, and thus 'revived' into a living legal tradition, by glossators who wrote commentaries in the side lines of the actual texts.

Building on Knuth's *Literate Programming* (1984) and modern applications such as *Jupyter Notebooks*, where developers can annotate their code with human-readable narrative and explanations, Krajewski proposes a more elaborate form of commentary on software code, "source code criticism". It should be underlined that this is something more far reaching than the commentaries of medieval glossators making Roman Law accessible and actionable for use. Krajewski's proposal is to create a new type of profession: code critics producing software commentaries in the tradition of critical humanities and enlightenment ideals of criticism, deconstruction, historization and discursivization to enhance digital literacy and agency.

In "listing 4" Krajewski exemplifies how such a critical commentary could look like. This commentary on a piece code that mimics the randomness of picking a card from a deck, not only explains the meaning of the different bits of code but is also a well referenced piece of academic work on the history of serendipity and randomness, and its literary reverberations. It is a beautiful piece of text that I have printed, framed and put on my wall as a textual work of art. While "listing 3" also provides a historicizing explanation (of which it is unclear to me if it would qualify as a basic form of source code criticism or not), it is clear that "listing 4" truly epitomizes the genre.

However, the question is if source code criticism can be more than beautiful scholarly work and a piece of textual art, and potentially provide an answer to the problematic opacity of artificial intelligence (AI) models, as Krajewski seems to suggest.

The world is increasingly populated by AI-fuelled systems. Transparency and interpretability, have been posited as the panacea against becoming subjected to the algorithmic opacity ("*computer says no*") and the unaccountability and uncontrollability (*the sorcerer's apprentice*) of such sys-

[46] Katja de Vries, 'GDPR as Hermeneutics' in *Common Erasures : Speaking back to GDPR* (ETHOS Lab 2020).

tems. However, transparency and interpretability are never goals in themselves, and when operationalizing them the question should always be: for whom and for what purpose?[47] Is it to empower a consumer, to inform a judge in a liability case, to make a manufacturer more self-reflective, or to motivate a decision towards a citizen?

To understand transparency in relation to AI-fuelled systems there are two ways of approaching them: either as *texts* or as *organisms*.

Let's start by looking at both law and software, in line with Krajewski's proposal, as texts. What are their parallels and divergences in terms of origin and destination?

In terms of authorship there is an interesting parallel between those writing legal texts and software: there is a certain impersonality, or as Savigny said in relation to the Roman jurists: "fungibility"[48], in the authorship of legal commentaries that also exists in those writing software code. Their output is more akin to a contribution to a growing coral reef than an individual literary work. Schiavone describes Roman jurists as having "a common awareness (...) that they were participating, with their own intelligence, decision after decision, writing after writing, in the collective formation of a grand ontological architecture."[49] When looking at the software that makes, for example, a contemporary mobile phone tick the biggest part of the software is no longer attributable to an individual author but is an accumulation of code that has undergone several cycles of being open-sourced and closed, and is constituted of endless adaptations, over-writings, and cuts-and-pastes.

However, when comparing legal texts with software code created using AI, that is machine learning (ML), methods, it is clear that the authorship of the latter is more indirect: an algorithm that creates a model from training data can be surprising or opaque to its developer in a way that a legal text would not. The origin of ML software is somewhere in between intentional human creation and evolutionary emergence.

In terms of readership there is also a clear divergence. The legal text always addresses a human reader. The totality of the system of Law is a legal fiction that only exists in its enactment by human lawyers, who put their legal creativity to work to construct arguments as to why the Law says one thing or another.[50] The law exists in its activation in legal narratives created by legal practitioners. In contrast, the primary addressee of software code is always a machine (or, to put it more precisely, the compiler that translates the code into binary executable code). The text works if the machine works. What is created is a system – or: a machinic organism – that can do something. Human understanding is important but secondary.

Transparency in relation to AI-systems as texts, that can be critically commented upon, is an Enlightenment dream from the era of books. Nobody can be against more digital literacy, and source code criticism can contribute to that, but AI-systems also need to be made transparent in a more actionable way: as organisms that can be challenged in interaction, counter-profiled[51], and questioned.

# References

Hildebrandt M, *Smart Technologies and the End(s) of Law. Novel entanglements of Law and Technology* (Paperback edition, Edward Elgar 2015).

Knuth DE, 'Literate Programming' (1984) 27 The Computer Journal 97.

Latour B, *The making of law: An ethnography of the Conseil d'Etat* (Polity 2010).

Schiavone A, *The Invention of Law in the West* (Belknap Press 2012).

Vries K de, 'GDPR as Hermeneutics' in *Common Erasures : Speaking back to GDPR* (ETHOS Lab 2020).

— 'Transparent Dreams (Are Made of This) : Counterfactuals as Transparency Tools in ADM' (2021) 8(1) Critical Analysis of Law.

[47] Katja de Vries, 'Transparent Dreams (Are Made of This) : Counterfactuals as Transparency Tools in ADM' (2021) 8(1) Critical Analysis of Law.

[48] Aldo Schiavone, *The Invention of Law in the West* (Belknap Press 2012) 8.

[49] ibid.

[50] Bruno Latour, *The making of law: An ethnography of the Conseil d'Etat* (Polity 2010).

[51] Mireille Hildebrandt, *Smart Technologies and the End(s) of Law. Novel entanglements of Law and Technology* (Paperback edition, Edward Elgar 2015).

# Author's reponse

**Markus Krajewski**

I am much grateful to Katja de Vries's inspiring comments on my approach on commenting code. I feel well understood. I'd like to comment her comments with three questions, in order to continue the lines of thoughts. Not being a legal scholar I prefer to comment on the coding part rather than on the law concepts.

### *Who writes code? (As a reply to who writes law?)*

Today, most of the code emerges as the effect of an interplay between one or more human authors, a vast collection of scripts which already exist and may have solved the questions at stake, and a specific writing environment (IDE) which offers help in researching documentation, examples of code snippets by others, as well as providing standard solutions generated by large language models (such as github Co-Pilot). The fact that in most of these processes a human author is still involved, and the fact that most of the code is written in - more or less - common English may serve as two indications that their development and algorithms as such are to be considered a social artifact. So, algorithms in most cased address humans rather than machines, in order to be traceable, understandable, and alterable. Otherwise algorithms could be written yet in Assembler or other arcane code (like the programming language 'brainfuck' which is so minimalistic that it aims at not being understandable at all). Considering these conditions of sofware production, it seems all the more urgent to keep the algorithms - which always already aim to a certain extent at humans - as transparent as possible at the source code level.

### *Whom to blame?*

In software development authors usually hide behind pseudonyms. As a part of a large, world-wide distributed community working on a specific code project like github, one communicates with a whole zoo of strange creatures, rarely announcing their real names, instead bearing awkward signifiers like 'commanderkotori' or 'jesuschrist', framed by generic profile icons. So who can be made responsible for a certain change of a line of code, who can be contacted if the author is opaque, not graspable since he/she/it (if 'it' is a bot) seems to be a secluded entity, hidden behind many veils. Shortly after its introduction, the git program which handles the versioning of widely distributed software developers' contributions, offered a specific function called 'git blame' which allows to identify an author who made a certain change to the collective code in order to reach out to him/her to discuss the change. With this feature, at least one veil of the different layers wrapped around a person's identity is removed and one can enter into a discussion with the responsible person. This adds, again, another social component to the entangled interplay between code and human actions in developing algorithms. And it undergirds the need to write the code already not only for the machine, but also for others to be able to grasp it. Otherwise, the coder will frequently get 'blamed',

### *Why is Source Code Criticism rather technology than art?*

Based on the two levels of meaning of the Greek term texne (art and technique), SCC aims at both levels equally: It is art in the closest connection with technology, or conversely an artful technique that helps to keep the path to understanding code and also AI open in particular. AI systems, however obscure and incomprehensible the large models may seem to even their developers and experts, are also built with computer science techniques, i.e. the scripts and blueprints are available as source codes that can be dealt with using the same methods of SCC which, thus, adds to the just emerging field of Critical AI Studies. To the extent that not only the algorithms but also the scripts for building the LLMs as well as an assessment of the underlying data sets are subject to criticism, the black box commonly called AI might lose its darkness. In this sense, ; also attempts to conduct a kind of educational program

against human immaturity in their relation to machines on
a very pragmatic, neither artificial nor artistic level.