# Scholarship in interaction: case studies at the intersection of codework and textual scholarship

Zundert, J.J. van

# Chapter 5

# Code, Scholarship, and Criticism: When Is Code Scholarship and When Is It Not?[1]

> "l'historien de demain sera programmeur ou il ne sera plus"
> (Le Roy Ladurie 1968)

## 5.1 The Softwarization of Scholarship

There is no single easy definition of code. Code can be regarded as a new semiotics with its own literacy (Knuth 1984; Vee 2013). It can also be seen as a mode of existence of software, which at least has two such modes: a textual and a processual dimension (Hiller 2015). The textual dimension is connected to code in its form of source code, which is the text produced by a programmer in a formal language that – once interpreted by a computer – results in executable software. The processual dimension is connected to the execution of code as a computer program, which invokes also a performative nature. Mark Marino has argued that we should "analyse and explicate code

[1] A previous version of this chapter has been published as Van Zundert, Joris J., and Ronald Haentjens Dekker. 2017. "Code, Scholarship, and Criticism: When Is Coding Scholarship and When Is It Not?" *Digital Scholarship in the Humanities* 32 (Suppl_1): i121–i123. https://doi.org/10.1093/llc/fqx006. This article was co-authored, but the greater majority of research and all of the writing was done by the first author, supported by comments from the co-author. Some sentences were redacted for clarification and a paragraph more precisely delineating "code criticism" in the context of this book was added.

as a text like any other, 'a sign system with its own rhetoric' and cultural embeddedness" (Marino 2006). For the argument presented here code is regarded as source code mostly. That is, code in its guise as blueprint for a program that can be executed. With regard to such code the question is when a particular piece of code acquires a scholarly nature. What properties or qualities force us to consider the source code of software as a scholarly object of study? And if we can determine those properties, then how do we evaluate the scholarly merit of these code objects? As we shall see in answering these questions, the operative aspect of code ("what it does") turns out to be of essence too.

However, before turning to such issues a pivotal question needs to be answered first: why does code deserve scholarly attention at all, for in past decades it has not been a given that code is indeed of scholarly interest (cf. for instance Bauer 2011). A rationale for the humanities to consider code as a scholarly object of study and to consider code as a scholarly object itself can be argued along two approaches at least. The first is related to a general "softwarization" of society as described by, inter alia, Berry (2014). The second is a more specific realization of this trend that relates to how we understand tools as instruments applied in research.

The "softwarization" of society that Berry argues has also been vividly described by Jones (2014) who refers to it as "eversion". This "eversion" is a term coined in the 2007 novel *Spook Country* by William Gibson, who is also famed as the author of the cyberpunk cult novel *Neuromancer*. The concept of "eversion" serves to identify the process of cyberspace turning itself inside out and flowing out into society beyond the point where either is truly separable (Jones 2014:28). Where prior to 2007, cyberspace was an alternate but separate and virtual reality into which human existence in some visions might eventually even transmigrate; after 2007, the ubiquity of access points to the digital realm, the omnipresence of embedded computer technology, and the primacy of digital streams as carriers of information let the worlds of the virtual and of reality merge and intersect to a point that it is very hard to tell them apart. Jones marks the appearance of the smartphone around 2007 as the point of articulation between these realizations of digitality. Berry describes in a similar vein the pervasiveness of computation and digital information, and questions it from a perspective of critical theory. At

this point in time, cultural artifacts and the processes of creation and interpretation tied to these artifacts are as much digital as they are not. Arguably therefore, the humanities should concern themselves with the humanistic status and interpretation of such artifacts and with the creative processes that they result from.

Concerns with how pervasive forms of computation affect society are raised often in the context or as a result of critical theory. People such as Coyne (1995), Berry (2014), Marino (2006), and McPherson (2012) approach the digital from a socio-philosophic vantage point and interrogate how social context shapes software and how it in turn affects society and the relation of humans to digital technology – mostly with the aim to critically examine whether the technology liberates or limits the potential for personal, cultural, or social freedom and development. The omnipresent and massive impact of digital objects and processes on society and culture should also be of concern to the humanities in and of itself because it deeply affects the socio-technical processes by which cultural artifacts are created and interpreted, thus affecting the object of study of the humanities.

There is also a more narrow methodological rationale for the study of code in the humanities. Just as software and digital information pervades society, it emerges in the humanities virtually everywhere. It appears both as source and object of study, e.g. in the form of digital data and information, and as resource, in the form of tools and infrastructure. If code is thus an emerging object *and* method of study – such as text is for the humanities – it should arguably be the subject of scholarly examination. A rejoinder to this is the often-invoked metaphor that one does not need to understand an engine to drive a car. That however is an improper metaphor for software. An article by Ian Hacking (1981) "Do We See Through a Microscope?" will be useful in understanding why this metaphor is erroneous, even pernicious. Hacking's argument centers on the question of how to establish the reality of what we see with a microscope. Fundamentally there is no way of knowing this. As humans we cannot empirically verify or testify that there is an object under the microscope when it is too small to sense. We trust however that the theory of optics holds, and that therefore the image we perceive is true to the nature of the object. We accept and trust that the way light passes through a system of lenses is accurately described and predicted by the theory of optics.

Yet this remains "just" a theory, despite the fact that it has repeatedly held up under testing. But exactly because no one has yet been able to prove that the theory is incorrect regarding the behavior of light in a microscope, we trust that what we see is what is actually there. Or in Hacking's words: "It may seem that any statement about what is seen with a microscope is theory-loaded; loaded with the theory of optics or other radiation. I disagree. One needs theory to make a microscope. You do not need theory to use one".

Hacking's remark sounds very similar indeed to "One needs computer literacy to make software. You do not need computer literacy to use it". The crucial difference is that code and software are not governed by a law of nature in the same way optics are. If the curvature of a lens is incorrect a user will get a foggy or blurred picture of a plant cell (for instance). But it will remain a blurred picture of a plant cell. No matter how broken the lens, it will not transform a picture of a plant cell into a picture of the facetted eye of an insect. Software code by contrast is "written" or "built" by humans and is not bound to natural rules of proper and verified behavior. Most mobile phones carry an inbuilt lens these days, with a camera "app" to take pictures. It would be rather easy to change the camera's software in such a way that whenever a user takes a picture, some random picture on the Internet would be presented as the photograph. Thus what Hacking justifiably concludes for microscopes on the basis of a general and well-supported theory of optics does not hold for software. In both cases there is a situation of trust. In the case of lenses we trust that a well-verified theory of light and optics will hold and that the nature of light and its interactions with materials will not change overnight. In the case of software there is a trust that the result of creative coding work will do what the creator of that work says it will do. But software tools are lenses of a different kind: at the time of writing according to TextMate (a robust no-nonsense text editor for Mac OS) this text up to here has 1,167 words, according to MS Word it has 1,174. If something as deceptively simple as counting the number of words in documents gives different results in different pieces of software, how do we trust complicated topic modeling software like Mallet that produces hundreds of clusters of terms as suggested topics found in a corpus? Software is governed not by laws of nature, but by the rules that are programmed into it by the engineer, that can be set by anyone having access to the design process of the software, and that

can result in incredibly complex heuristics and algorithms. This fact should by itself warrant some systematic approach to critiquing code. But especially now that more digital tools are getting integrated into the methodology of humanities, the adequacy and validity of analyses depend to a certain extent on an adequate understanding of such specific rules.

## 5.2 Scholarly Assumptions in Code

To make this more concrete let us study the case of CollateX (Haentjens Dekker et al. 2015). CollateX is software under active development at the Huygens Institute for the History of the Netherlands.[2] CollateX is – as the name suggests – a collation engine. The core of CollateX consists of an algorithm – that is, a defined sequence of precisely specified steps that produce an output (Schmidt 2016). Algorithms as mathematical and programming concepts have a long history of themselves (cf. Bullynck 2016). Also some inroads toward the study of algorithms have been made from the humanities and social sciences, most noticeably from the perspective whether and what knowledge of algorithms is pertinent to humanities (Seaver 2013; Schmidt 2016). Here I am not interested so much in a mathematical proof of CollateX's working, but in a similar vein as Seaver we want to know how particular assumptions of the developers about text and text scholarship become inscribed in the algorithm that makes up the core of CollateX. In the following all statements on the CollateX software pertain to the 2.0.0 version of the Python port available on the Python library repository PyPI (Python Package Index).[3]. The open-source code of CollateX is available under GPLv3 license in GitHub.[4]

CollateX's algorithm, if given a number of texts that are largely but not exactly the same, will align the parts of texts that run parallel, or "match" as this is usually called. For instance, if the algorithm is given the following texts:

```
1. the black cat hops over the red dog
```

[2] http://collatex.net/

[3] https://pypi.python.org/pypi/collatex

[4] https://github.com/interedition/collatex/tree/master/collatex-pythonport

```
2. the white cat hops over the dog
3. the black cat hops over the red cat
```

It would align these "witnesses" (as variant texts are usually called in textual scholarship) as follows:

```
1. the | black | cat hops over the | red | dog
2. the | white | cat hops over the |  –  | dog
3. the | black | cat hops over the | red | cat
```

Collation is a scholarly task central to the field of textual scholarship, which is itself concerned with establishing a solidly argued representation of a given text. Because the process of collation is labor-intensive, repetitive, tedious, and error-prone (Robinson 1989), it is a good candidate for automation. As with all software, any such automation will result in an implementation of an algorithm that to a certain extent rests on particular assumptions (Lehman and Ramil 2000). The algorithm of CollateX makes three tacit assumptions on the heuristics of alignment:

1. It is desirable to minimize the number of differences between witnesses
2. Phenomena that are shared across most witnesses should be preserved
3. The number and order of witnesses are arbitrary

Furthermore the algorithm of CollateX is based on at least one axiom that states that it is computationally infeasible to distinguish between a transposition and a combination of substitution and deletion. That is, if the algorithm finds the following alignment:

```
1. the cat hops over the black dog
2. the dog hops over the black cat
```

It is nigh impossible for any computational algorithm to decide whether the "cat" and "dog" in the first sentence were switched (textual scholars speak of

a "transposition") or if either of them was individually replaced (i.e. substituted by a consecutive deletion and addition).

The issue here is not whether these assumptions are correct, but rather that they exist in the code as such. They represent rules and choices that could have been different as a result of different scholarly reasoning and argument. Assumptions are inscribed tacitly in code rather than being explicitly mentioned or described by it. It would be very hard indeed, even for skilled engineers, to reverse engineer or read the code so that these assumptions become apparent. Yet they are part of the very rationale behind the mechanism that fulfills the scholarly task of alignment.

In the case of CollateX, the aforementioned assumptions may not be shared by each textual scholar. They are indeed not laws of nature, nor are they generic mathematically proven principles. Especially the axiom concerning transpositions could be subject to scholarly debate. A human reader will apprehend quickly that in the example above, the "cat" and the "dog" were transposed. But unless evidence external to the texts is presented, fundamentally this is not deducible with complete certainty – it could have been that the cat was replaced with another dog. The apprehension of the human reader is in fact an assumption, conjecture based on intuition and experience. A rule of thumb could be that when more words are involved in a potential transposition (so longer fragments are switched) and the fewer words there are between the two potentially transposed fragments, the likelier it is that a deliberate transposition occurred. It is unlikely that an author would for instance switch around a "the" at the beginning of a text with a "the" at the end of that text. If we find "It was a dark and stormy night" in one witness at the beginning of a text, and in another witness at the end, it is more likely that deliberate transposition was the cause. It would be very time-consuming to take this rule of thumb into account when computing the alignment of witnesses because the number of comparisons that need to be performed by the code would grow exponentially. Hence the axiom: it is fundamentally impossible to know from the texts alone if a transposition happened, and it is computationally highly costly to compute all potential transpositions; thus, it is computationally infeasible to distinguish between a transposition or two independent substitutions.

The third assumption, which posits that the alignment should be independent of the number and order of witnesses, is also debatable from the perspective of textual scholarship. Suppose that it is clear from external evidence – e.g. from the bindings of a manuscript or the type of materials used – that a particular witness is older than any other. In those circumstances it becomes a legitimate scholarly question whether that witness should be a guiding text, or a "base text" as it is called when specifically used as a guide for decision-making in the process of alignment (Roelli 2015). In unmarked situations, however, it is assumed that baseless collation is preferable (cf. Andrews and Macé 2013). During the development of CollateX, great care was taken therefore to prevent it from presenting a result that is in some ways biased or colored by the particulars of one specific witness. Indeed this feature became a "unique selling point".

The contention based on the above is that code through its mathematical and algorithmic origins does not acquire some inherent objective and neutral correctness. Instead the construction of code is situated and depends on the assumptions of its builders, be they subjective, supposedly objectified, or scholarly. In this respect code and software cannot escape what has been similarly found for data and facts. There is no such thing as "raw data" (Gitelman 2013), rather data and facts are carefully constructed (Bowker 2006; Latour and Woolgar 1986). This does not deny the potential solidity of facts, but it calls attention to the situatedness of their creation. Even what "constitutes" data is dependent on context and often even far from clear, especially in the humanities (Borgman 2015; Kouw, Van den Heuvel, and Scharnhorst 2013). Within the digital humanities this has given rise to criticism on how data should be understood, on data representation (Drucker 2011), and on the use of (standards for) digital formats (Vitali 2016).

## 5.3 Scholarly Code Criticism

The assumptions that underpin the code of specific software in textual scholarship ought not to be the idiosyncratic musings and intuitions of individual programmers. In the case of CollateX assumptions were inferred from close and repeated conversations between the lead developer and a variety of

textual scholars who had a particular interest and experience with text collation. These assumptions are in this sense a result of aggregated, carefully interpreted scholarly knowledge re-inscribed in code. It is this process of aggregation, interpretation, and re-inscription of knowledge that lends the code of CollateX a particular scholarly nature. Insofar as interfaces and code bases can also be thought of as arguments (cf. Galey and Ruecker 2010), it is these assumptions by which the code of CollateX captures and adds to the ongoing scholarly debate on collation. As argued above however, the argument that code makes is very implicit. How can scholars – or for that matter other programmers – examine and critique this code and these assumptions as an integral part of academic discourse?

This question points to a clear need for a method or a framework within the humanities to systematically explore and validate scientific software engineered for and used in the humanities. No such agreed upon formal method or framework for critical evaluation of code exists. Nor is there an agreed upon method to share any results of the critical evaluation of code. As Mark Marino has stated in a field report on critical code studies (CCS): "there remains a considerable amount of work to develop the frameworks for discussing code" (Marino 2014). Marino's report presents a concise history of CCS that suggests that they are indeed an application of critical theory. CCS studies the social context and processes surrounding code and its creation. A good example is McPherson's 2012 contribution to *Debates in Digital Humanities*, titled "Why Are the Digital Humanities So White? or Thinking the Histories of Race and Computation" (McPherson 2012). Read superficially it is an article that will make many (white male) computer engineers roll their eyes and sigh: sure, UNIX is racist. However that is not McPherson's argument: "I am not arguing that the programmers creating UNIX at Bell Labs and in Berkeley were consciously encoding new modes of racism and racial understanding into digital systems. [...] Rather, I am highlighting the ways in which the organization of information and capital in the 1960s powerfully responds – across many registers – to the struggles for racial justice and democracy that so categorized the United States at the time. [...] The emergence of covert racism and its rhetoric of color blindness are not so much intentional as systemic. Computation is a primary delivery method of these new systems, and it seems at best naive to imagine that cultural and

computational operating systems don't mutually infect one another."

Another clear concern of CCS is the aesthetics of code and code-as-text. Marino (2006) is interested in reading code as text: "I would like to propose that we no longer speak of the code as a text in metaphorical terms, but that we begin to analyze and explicate code as a text, as a sign system with its own rhetoric, as verbal communication that possesses significance in excess of its functional utility." Given this proposition it is understandable that CCS is fascinated with poststructuralism-inspired uses and interpretations of code, such as Alan Sondheim's concept of codework that mixes computer code and text, and in which computer code thus additionally becomes a medium for artistic expression (Wark 2001).

Although critical theory inspired code criticism arguably should be part of any framework for evaluating the scholarly qualities of code in the humanities, the approaches and examples from the field of CCS also still leave a lot to be desired. To understand this, consider a remark Evan Buswell made during a HASTAC 2011 CCS event (Marino 2014). Buswell stated that CCS cannot only deal with the arbitrary elements of code because that would relegate code criticism to aesthetics only. This was a reaction to Mark Marino's suggestion to try to read code as text and to use code variables as meaning forming elements to see how this would give expression to the meaning of code. Buswell was quick to note that variable names are arbitrary because of an ubiquitous code mechanism called indirection. Variable names are wrappers and boxes: what is printed on them needs not to have an intrinsic relation with what is in them. Thus if one reads in e.g. JavaScript:

```
var welcome_message = 'Welcome to my homepage!';
```

It simply means that there is a variable with the name "welcome_message" that holds the text "Welcome to my homepage!". However, that name is arbitrary. The code:

```
var bananas = 'Welcome to my homepage!';
```

creates the same result (which is that there is a variable with the text value "Welcome to my homepage!"). Thus the name of the variable does not entail anything about the value of the variable or its meaning within the code, or beyond.

Mark Marino's argument was based on the assumption that developers usually use "speaking names" for variables, precisely because it keeps the code somewhat readable, and hopefully clear to other developers. Under these conditions variable names may indeed reveal something about the assumptions and norms connected to the context in which the code was developed. If the variable was named "opening_sentence" instead of "welcome_message", this may reveal something about the intention or frame of mind of the developer. The former might indicate an engineer foremost focused on text structure, the latter might suggest that the programmer was thinking more about user interaction.

Thus there is certainly reason to do as Marino suggests and to read code also simply as "a text". However, code is a text that performs. It also represents a program that can be executed, and fundamentally variable names do not reveal this performativity. They do not reveal necessarily the aim of the code, nor how it operates. Thus, as Buswell concluded, student engineers may learn from CCS to carefully choose their variable names because they will be working with culturally sensitive programmers in various cultural contexts and settings – but "all the while there will be an invisible line between CCS and CS, protecting the core from the periphery, insulating and separating from critique the power structure of code itself, and constructing a discourse of good code and bad code to go along with the discourse of good business and bad business that tends to dominate naive anti-capitalist critique".

Before I claim that a solid framework for criticism of source code applied in scholarship is lacking, as I will, I should explain what this means exactly. Of course there is a tremendous amount of work done in both software studies and media studies to construct frameworks to critique software. Lev Manovich's *Software Takes Command* (2013), *Expressive Processing* by Noah Wardrip Fruin (2009), and the already mention work by Tara McPherson (2012) testify to this achievement. But to call these works "code criticism" would be somewhat of a misnomer. And in fact that is not where these scholars position themselves: they talk about software and media criticism. These works study the effects of software, which is the performance aspect of code: that what the user sees, the interface, its uses and affordances. They study the performative aspect of software as a performance: how software creates effects in people's behavior, work, opinions, and so forth. They also study soft-

ware "in the wild": the socio-political effects of mainstream software used in industry, as games, and as a tool of personal and institutional productivity. But what they do not do, is critique or evaluate the actual source code of software, the particular kind of text that formulates the behavior of software. Also their object of study is that of software in society, not specifically that of source code applied as a scientific tool in the humanities (or sciences). Hence they are not studies of scholarly code and they are not code peer review, but they are studies of social effects of software at large.

This is what I mean when I speak of "code criticism" in particular: a scientific framework for peer review of source code that is written specifically for and applied in textual scholarship (or other scientific fields) to evaluate that source code for its scientific reliability, implied methodological choices, and implicit scholarly interpretations. My concern is the fact that we increasingly use bespoke code – i.e. tailor made, one off applications to serve specific purposes and specific concerns in particular scholarly editing or research. For the critique of this kind of code, in textual scholarship and by implication the humanities more in general, we lack an established theoretical and practical framework.

As a framework for code criticism CCS seem to lack a rigorous method for examining and critically interrogating actual code beyond reading the "code as text". In addressing this it would make sense to draw a parallel between the interdependent relationship of textual criticism and literary criticism on the one hand and between code criticism and CCS on the other hand. Literary criticism is the application of critical theory and aesthetics to literature. It is occupied with the interpretation of literature, its contextualized meaning, its cultural inwardly and outwardly influences, its development over time, etc. Textual criticism is less about reception, meaning, cultural situatedness, and writerly text.[5] Rather it is the critical skill of establishing a well-argued representation of a text. Though "fact" in the light of post-structuralist theory is a problematic term to say the least, it is not unreasonable to posit that textual criticism is pre-occupied with scientific textual fact finding and accountability: textual criticism tries to establish as close a "factual" representation as

[5]For a concise explanation of "writerly text" see Mambrol (2016). This aspect is also dealt with in some more depth in the next chapter.

possible of a text through a scientifically accountable process (cf. McGann 2013).

Textual criticism faces its own particular challenges resulting from digitality. Since authors turn to personal computers and text processing software for text production, textual criticism – used to an almost exclusively physical materiality of manuscript and print publications – is confronted with the realities of digital materiality. Scholars in this field are therefore augmenting and adapting existing methodologies to this new reality. This includes, for instance, new approaches to the preservation of personal archives left by authors on hard drives (Grigar et al. 2009; Kirschenbaum et al. 2009). It also includes adaptation of scholarly methodology aimed at studying the genesis of authorial documents, since genetic stages change from "manuscript draft" and "print proof" to revisions stored in for instance .docx files, the standard file format for more recent versions of MS Word (Ries 2010; Buschenhenke 2016).

Arguably a framework for scholarly evaluation of code could encompass components of CCS and components that are more directly aimed at factual code review – similar to how text critique encompasses literary criticism and text criticism. The CCS component would focus on answering questions of broader socio-technical impact. For instance, is there an ideology underlying this code? What are the cultural assumptions and biases apparent in the code? What was the social context of its development? The code criticism component would aim at critically examining the actual code and its scholarly or scientific intentions. What is the stated purpose of this code? Which scholarly task – perhaps in relation to the concept of scholarly primitives (Unsworth 2000) – is it trying to accomplish? How well is it accomplishing that task? What concepts and relations are modeled into the code?

Code criticism in this sense would first of all be pragmatic. If literary criticism asks the question "What does this mean?" and CCS ask "How does this code affect us?", then textual criticism asks "What was written here?" and code criticism asks "What does this code do?" Code criticism could deliberately pose deceptively simple questions to code because this aids in revealing the

scholarly nature of code. As an example one can compare CollateX with eLaborate, another tool developed for use by textual scholars.

ELaborate is a tool for digital transcription created and actively maintained by the Huygens Institute for the History of the Netherlands.[6] Transcription is undeniably a scientifically valid and valuable primitive of humanities, especially with regard to scholarly editing and philology. Is therefore eLaborate to be deemed a scholarly tool? The software supports the scholarly task of transcription. Does this mean that the software and the code itself are scholarly and thus examples of scholarship? The key is in the distinction between enabling and performing tasks. ELaborate enables the scholarly task of transcription, but the transcription itself and all the scholarly skills and decisions tied to it are still performed by the user. ELaborate is not somehow magically more adequate in registering the keystrokes of a scholarly editor than WordPress, MS Word, TextMate, or any other text editor. It has a number of features that greatly facilitate the task, and allow the editor to really focus on it. Otherwise it does its best to get as much out of the way of the scholarly editor as it can. It has less feature clutter than for instance Word, it has a centralized and institutionally backed repository for all its data, it is Web-based, and so forth. In comparison with other tools this means that there is seemingly always one specific feature that makes eLaborate a better fit for the scholarly task than most other text editors. Yet it would be hard to argue that the code propelling eLaborate is scholarly in itself and by itself.

This distinction of the scholarly nature of code is based on the question whether "scholarly decisions and choices are delegated to the code level". This implies that there is no absolute certain measure that can tell whether code is scholarship because establishing the scholarly nature of the code depends on convincingly arguing that such decisions indeed were delegated to the level of code, which is an argument that always should take into account the situatedness of both the code's development and its intended purpose. In the case of CollateX this can indeed be argued considering the currently pervasive practice of aligning variant texts by hand in textual scholarship. Decisions currently normally taken by a scholar are delegated to the code, as the algorithm results in a possible alignment of variant texts

[6] http://elaborate.huygens.knaw.nl/

that implies decisions on which words between variants match. In contrast, the code of eLaborate does not effect or propose as a result such choices or decisions that currently would be deemed scholarly significant in textual scholarship.

In all, this then does not preclude at all that the "code building" connected to the development of eLaborate can be a scholarly valuable act or contribution. The development of eLaborate is certainly a valuable scholarly achievement: scholarly thought and argument were part of the process of its creation and the design of its specific functionalities (Beaulieu, Van Dalen-Oskam, and Van Zundert 2012), and much subsequent scholarship was enabled through the use of eLaborate. Because scholarly argument at some level is involved, it is still relevant to critically examine eLaborate's interface, features, and capabilities. This would be tool criticism however, not code criticism. Obviously tool criticism might at some point very well be integrated into the approach suggested here, but that is beyond the scope of this argument. In the case of CollateX decisions that are understood as scholarly responsibilities in the current context of textual scholarship are delegated more extensively to the code itself than in the case of eLaborate. Therefore, unlike eLaborate, the code of CollateX "performs" a scholarly task: based on the tacit assumptions built into its code, the algorithm of CollateX independently makes scholarly informed decisions – or rather proposes these, as the decisions until now are always ultimately corroborated by a scholar. It is arguable therefore that the code of CollateX in the current context of textual scholarship is endowed more with a scholarly nature than the code for eLaborate. That in fact the code of CollateX represents scholarship and is itself a scholarly object. This is no different from a monograph or print edition, each one a scholarly object whose scholarly nature arises from the arguments they constitute and represent.

Critically examining this argument and the scholarly nature of the code itself is not straightforward however. Above the mostly tacit nature of scholarly assumptions built into code was already pointed out. But code is unintentionally covert in other ways as well. Engineers often talk about the "model" that underlies their code. "Model" is rather a "hopelessly polysemous" word, as Willard McCarty remarks (2005:27). There is extensive literature meanwhile also in the field of digital humanities on the meaning, purpose, appli-

cations, and epistemology of models and modeling pertaining to scholarship, data, and code, notably McCarty (2005), Flanders (2012), Jannidis and Flanders (2013), Ciula and Marras (2016) – but there are many more. Obviously there is a relation between the analytic model and data used by researchers and the data and object models constructed through code by programmers. The collection of digital object (definitions) and their relations expressed in code make up the domain model, as defined by Fowler (2002:116), which essentially expresses the programmer's "understanding" of the models used by the researcher(s).

In the context of source code creation by software engineers, the model component of the code is thus that which comes to represent the conceptual or phenomenological model of the problem domain. That is, the concepts, the relations, and the operations that mimic the problems, objects, and processes the software developers are trying to automate or solve on behalf of a client or, in our case, a researcher. In the case of eLaborate, the model has coded objects such as "Transcription" and "Annotation". Annotation objects in the code may have associated functions or methods, such as "create", "update", or "delete". Of course all the components are needed in a meticulously orchestrated combination to make the software function; all components are in that sense essential to it. Not any framework for code criticism can therefore conveniently eschew some part of a body of code. However, the components that capture the domain model are probably the most closely associated with inscribing the conceptual model of the researcher into code, as opposed to data storage components or visualization components. To complicate matters even more maybe, visualization obviously constitutes a transformation of the data that by itself is modeled too. Visualization transformations therefore also constitute an interpretation and argument about data. Like tool criticism however, the problem of interface critique is out of scope here – even though it is readily imaginable that the "code" that drives visualizations could be subject to code criticism within a framework of code criticism framework.

Even if we boil code criticism down to critiquing the domain model, effective code criticism may turn out to be a strenuous activity. Domain models may be hard to gauge or peruse from the code that is eventually published, and they can be unintentionally obfuscated. Models may be as tacitly expressed

in the code as the assumptions underpinning it, or they may be confusingly cloaked by different code expressions. Part of the algorithm of CollateX, for instance, is based on a decision tree. This tree is used to recall which decisions were made by the algorithm to come to an alignment between witnesses. If a new witness needs to be added into the comparison, previous alignment solutions can be compared to favor one solution. For reasons of performance (i.e. speed) and scalability, the decision tree is not expressed in the code as a tree, however. Instead the engineer chose to use a matrix that will deliver the same power of decision but at a very much lower performance penalty. Reading directly from the code, it would be hard, or at least considerably confusing, to see that a matrix was used to perform the function of a decision tree.

Thus just as with the variable names that can be arbitrarily chosen and thus obfuscating, code may be for good reasons unintentionally enigmatic. The nature of code in this sense seems to resemble poetry more than prose. Poetry sometimes intentionally uses enigmatic or hermetic language, forcing the reader to reread and rethink possible meanings. Code will in general be less intentionally enigmatic, but will sometimes be no less hermetic. Sometimes such hermetic code becomes a goal in itself, for instance when coders try to come up with "oneliners": tiny algorithms of one line of code that perform certain, sometimes incredibly complex tasks. Arguably one of the best-known examples of this "onelining" is "10 PRINT CHR$(205.5 þ RND(1));: GOTO 10", to which even a full book publication was dedicated (Montfort et al. 2012). Such witty solutions may earn particular admiration of other coders, the solution being regarded as a particular "elegant" one. Yet the "coolness" of the solution may result in code that is particularly obfuscated and hard to read. Also the actual algorithm may be counter-intuitive even if mathematically highly efficient. This is arguably the case with the Quicksort algorithm, which is an algorithm for sorting that carves the series to be sorted into subseries.[7] This "divide and conquer" strategy results in a performance gain (i.e. sorting speed) several magnitudes larger than the approach generally found to be more "intuitive", called insertion sort, which involves starting at the top of the list and inserting each item in the series in its "nat-

[7] https://en.wikipedia.org/wiki/Quicksort

ural order" position.[8]

## 5.4 Criticism in a Continuum of Literacies

How then do we critically examine code that may be particularly hard to read, scrutinize, and understand? At the very least, an attempt should be made at reading the code, even if simply to establish the degree of readability of the code, because this is valuable information for criticism too. If the code is nigh incomprehensible, what does this mean? Can the reasons for possible intentional obfuscation be deduced and/or reasonably established? Is the illegibility a result of unskilled coding? Obviously inline comments and external documentation should offer help in determining the intent of the code as well. Also establishing the software development methodology used can reveal useful insights. There are various methodologies to build software, from highly formalized and rigorous to fully pragmatic "cowboy coding". Some methodologies are bound to be a better fit than others for the heterogeneous nature of humanities data and research questions (Van Zundert 2012).

Mostly however: why not talk to the creators of the code themselves? Assuming that engineers indeed apply current so-called "good practices", software development is a highly dialectic practice. The adequacy and effectiveness of code are mostly determined by how well the model that is inscribed in the code fits the domain model of the problem or task that the software was developed for. To deduce a best-fit model, engineers should go to great lengths. Analysis and design for modeling in most current software development methodologies will involve deep client and/or user interaction. That is, during the design phase, engineers will interview the client over and over again to explore the exact properties of the domain model. And during any implementation phase, engineers will in all likelihood repeatedly expose the execution of the code to the scrutiny of the researcher and will adapt the design iteratively to what the researcher reports back as to shortcomings, omissions, etc. Thus the model is designed, tweaked, and tuned in a continu-

[8] Cf. https://en.wikipedia.org/wiki/Insertion_sort

ous communicative and dialectic feedback cycle between developers and researchers.

If the engineering of a model is governed by dialectic, the most adequate mode of scholarly code criticism could be parallel. Code as an argument can be adequate but obscure, and in such cases a good way of establishing the model tacitly underlying the code could be to reverse engineer it through discourse. Thus by reversing the dynamic of the dialogue, we may understand software in the same way as its development was articulated and argued: by a deep and continuous, even "intimate" as Frabetti (2012) suggests, dialectic. What is reversed during the phases of creation and criticism is the role of the interviewer and interviewee.

A similar parallelism and mirroring arises in another potential avenue for critically examining code. It is a good practice in code engineering to develop not just code but also a test suite for that code.[9] A test suite or harness is a set of tests expressed as code that can be run to check that software is working correctly. Engineers can in this way guarantee the correct working of the code. Tests are used to check the workflow, to test against critical conditions, to inspect certain expected output for given input, to test the formal constraints of a model, and so forth. It may turn out to be as valuable for code criticism to examine the test suites that accompany code as the contents of the code itself. Much may be gauged from these tests about assumptions, corner cases, conditions, flow, limitations, and intent of the code.

But an even more intriguing application of test suites might be for scholarly code critics to develop these suites themselves. Currently test suites and automated tests for software are tools of the engineer. But there is no reason why the frameworks that help engineers to control, check, and validate their work would not be used to probe, explore, and test the same software by code critics. Instead of facing the engineer, test harnesses might just as well face the critic and user. Several people involved with CCS have expressed similar ideas. Montfort et al. (2012:322) speak of studying "software by coding new software". David Berry refers to such possible test suites as "coping tests" (Berry 2014).

[9] https://en.wikipedia.org/wiki/Best_coding_practices

The possible application of code to test code, to create test suites to examine codebases as a form of humanities informed criticism, can also be cast as a continuum of two literacies. Three decades ago Donald Knuth believed that the time was "ripe for significantly better documentation of programs, and that we can best achieve this by considering programs to be works of literature" (Knuth 1984). His language called WEB lets the same program produce working code as well as an explanatory narrative about that code. WEB however, did not find a broad audience, neither in computer science nor in the humanities – with the odd exception (e.g. Huitfeldt and Sperberg-McQueen 2008). Knuth was interested in code as a form of literature and in writing software as a specific kind of literacy. In other words, he was interested in how two kinds of literacy, that of computer language and that of human-authored text, could merge. As a proponent of computational methods in the humanities, Franco Moretti appears to agree that this merging has a lot of promise (Dinsmann 2016). Literacy enables one to write *and* read, to express *and* inquire. From these perspectives the understanding of the literacy of code and the literacy of text as different and opposed literacies seems artificial and intellectually lazy. Instead, to develop a valid and adequate mode for scholarly criticism of code, they need to be understood as variations within a continuum of literacies (cf. Kittler 1993b).

These notions so far are theoretical. Also in the realm of this chapter, I have only been able to sketch the outlines of a possible practical approach to code criticism for code created in the humanities. Based on these more theoretical ideas currently work is undertaken to explore how text and computational literacy may amalgamate into a concrete method for code criticism. This work takes the form of an iPython Notebook in development[10] that reports on the investigation of the code underpinning a publication by Ted Underwood and Jordan Sellers (Underwood and Sellers 2016). A follow-up publication will be dedicated to this explorative practical code criticism case study.

[10] https://github.com/interedition/paceofchange

## 5.5 Conclusion

Code criticism and code peer review are hardly even nascent in the humanities and digital humanities. Some work has been done in the realm of CCS, but these fledgling approaches have focused primarily outward from code and have considered code mostly as a culturally situated part of a larger sociotechnical system. Almost no examples of thorough code criticism exist that regard code from a humanities methodological point of view. The type of criticism that asks: what is methodologically expressed here, how is it argued, and how can we validate it? Given the large ramifications that digital information and software have for humanities sources, resources, and methodology, this situation is rather surprising, and methodologically unhealthy. In this chapter I sketched the outlines of an approach that would do justice to the work that has been done in the realm of code criticism but that would also self-reflectively turn criticism toward the code that promises new tools to the humanities.

For centuries, argument, logic, interpretation, and reason have been both the means to put forward results in the humanities as well as the tools to judge those results. Humanities methodology is highly self-reflective. Methodology now increasingly means digital methodology, but that does not imply that critical self-reflectivity should disappear: there is no self-evident correctness of technology just because it is digital technology. Much work still needs to be done to remediate the critical aspects of humanities scholarship into the digital realm. This is a critical task digital humanists should be aware of.