Code, Scholarship, and Criticism: When is Coding Scholarship and When is it Not?

Joris J. van Zundert (corresponding author)

Huygens Institute for the History of the Netherlands

Royal Netherlands Academy of Arts and Sciences

The Hague, The Netherlands

joris.van.zundert@huygens.knaw.nl

Ronald Haentjens-Dekker

Huygens Institute for the History of the Netherlands

Royal Netherlands Academy of Arts and Sciences

The Hague, The Netherlands

ronald.dekker@huygens.knaw.nl

Abstract

What is the scholarly nature of code and how do we evaluate the scholarship involved with coding? Our claim is that the humanities need an urgent answer to these questions given the increasing softwarization of both society and scholarship that pushes the boundaries of the methods and objects of study of the humanities. We argue that as a result there is a need to develop code criticism as a critical and reflexive tool within the humanities. Code criticism is described and positioned with respect to Critical Code Studies, textual criticism, literary criticism, tool and interface critique. Finally we outline an approach to code criticism based on ideas of reciprocal inquiry and of a continuum of literacies that connects code, code criticism, textual criticism, and literature.

The Softwarization of Scholarship

When does a particular piece of code or some code object acquire 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? These and similar questions as well as some of the potential answers to them are what we want to consider in this contribution. However, before we turn to these questions, foremost we need to answer the question why code deserves scholarly attention at all, as in past decades it has not been a given at all that code is indeed of scholarly interest (cf. for instance Bauer 2011). We contend that there are at least two approaches towards arguing that there is a rationale for the humanities to consider code as a scholarly object of study and to consider code as a scholarly object itself. The first is related to a general *softwarization* of society as described by 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 for has also been vividly described by Steven E. Jones (2014) who refers to it as *eversion*. This *eversion* is a term coined in the 2007 novel *Spook County* by William Gibson, who is otherwise known 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 then cultural artifacts and the processes of creation and interpretation tied to these artifacts are as much digital as they are not. Arguably therefore, humanities should concern itself with the humanistic status and interpretation of such artifacts and the creative processes 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 Richard Coyne (1995), David Berry (2014), Mark Marino (2006), and Tara 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. This omnipresent and massive impact of the digital 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.

We argue however that 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. In rejoinder to this the metaphor is often invoked that one does not need to understand an engine to drive a car. That however is an improper metaphor for software. To understand why, we refer to an article by Ian Hacking published in 1981: 'Do We See Through a Microscope?' 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 my writing according to TextMate (a simple 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.

Scholarly Assumptions in Software

To make this more concrete let us study the case of CollateX (http://collatex.net/), a piece of software under active development at the Huygens Institute for the History of the Netherlands (https://www.huygens.knaw.nl/). CollateX is—as the name suggests—a collation engine. It is essentially an algorithm that, 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) 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, 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 2000). The current algorithm of CollateX[1] 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 is 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 point here is not whether these assumptions are correct, but rather that they exist at all. 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 assumptions, expressed not so much in the code as through its performance, 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 our example above the *cat* and the *dog* were transposed. But unless evidence external to the texts shows up, 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. 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 the oldest. In those circumstances it becomes a legitimate scholarly question whether or not 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 & 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.

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. We would argue that 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 & Ruecker 2010; Van Zundert 2015) 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?

To us this suggests that there is a 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 code and the social context and processes that give rise to particular forms of code. A good example is Tara 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 makes 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 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 we are certainly convinced that code criticism from a critical theory approach should be part of any framework for evaluating the scholarly qualities of code in the humanities, the approaches and examples from the field of critical code studies leave something to be desired for. Our criticism runs parallel to 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 text as code and use variables as meaning forming elements and to see how this would give expression to the meaning of code. Buswell was quick to note that variable names are—through the for information technology pivotal technique of indirection—arbitrary in code. Variable names are wrappers and boxes: what is printed on them needs not to have an intrinsic relation 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.

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 tell us 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 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.”

As a framework for code criticism then, critical code studies seems 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 critical code studies 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[2] text. Rather it is the critical skill of establishing a well-argued representation of a text. Though 'fact' in the light of poststructuralist 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 to a 'factual' representation as possible of a text through a scientifically accountable process (cf. McGann 2013).

Arguably a framework for scholarly evaluation of code should in a similar vein encompass both components of critical code studies and components that are more directly aimed at factual code review. The critical code studies component would focus on answering questions of broader socio-technical impact. 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 is first of all pragmatic. If literary criticism asks the question 'What does this mean?' and critical code studies 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 deliberately poses deceptively simple questions to code, because this helps to reveal the scholarly

status of code. As an example we can compare CollateX with eLaborate, another tool developed for the use of textual scholars.

Elaborate: what does it do, and how does it do that? Elaborate is a tool for digital transcription created and actively maintained by the Huygens Institute for the History of the Netherlands (http://elaborate.huygens.knaw.nl/). Transcription is undeniably a scientifically valid and valuable primitive of humanities, especially with regard to scholarly editing and philology. Do we therefore deem eLaborate to be 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, 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. Still eLaborate is in some sense a scholarly achievement: scholarly thought and argument was part of the process of its creation and the design of its specific functionalities (Beaulieu et al. 2012).[3]

Unlike eLaborate, CollateX *performs* a scholarly task. Based on the tacit assumptions built into the code the algorithm of CollateX takes a number of scholarly decisions, which essentially decide how multiple text witnesses should be aligned. Scholarly responsibilities are handed off more extensively to the code in this case. We can therefore argue that the code itself has more of a scholarly nature than the code for eLaborate. That in fact the code 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. We have already pointed to the mostly tacit nature of the scholarly assumptions built into code. But code is unintentionally covert in other ways as well. Engineers often talk about the *model* that underlies their code.

Mostly the model component of the code is 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 problem, 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 then, 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. Any framework for code criticism cannot therefor eschew part of some 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.[4]

Critiquing the domain model, or even perusing it from the code can be hard as it may be unintentionally obfuscated. It may be as tacitly expressed in the code as the assumptions underpinning it, or it may be confusingly cloaked by a different expression. 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 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, such as when coders try to come up with *one-liners*: tiny algorithms of one line of code that perform certain—sometimes incredibly—complex tasks. Arguably one of the best known examples is '10 PRINT CHR$(205.5+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 particular obfuscated and hard to read, and the actual algorithm may also be counter intuitive yet mathematically highly efficient, such as in the case of the Quicksort algorithm (https://en.wikipedia.org/wiki/Quicksort).

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 intention 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 good practices, software development is a highly dialectic practice. The adequacy and effectiveness of code is 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 continuous 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. As an argument code may be adequate but obscure. 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 mirrored 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. 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 intentions 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 critical code studies have expressed similar ideas. Nick 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).

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 WEB language, which lets the same program produce working code as well as an explanatory narrative about that code, did not find a broad audience—neither in computer science nor in the humanities (with the odd exception, e.g. Huitfeldt & 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. Literacy enables one to write *and* read, to express *and* inquire. We argue that there is a great need to discard the understanding of the literacy of code and the literacy of literature as different and opposed worlds. Instead, to develop any valid and adequate mode for scholarly criticism of code, they need to be understood as variations within a continuum of literacy (cf. Kittler 1993).

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 Critical Code Studies, but these fledgling approaches have focused primarily outward from code and have considered code mostly as a culturally-situated part of a larger socio-technical system. Almost no examples of thorough code criticism exist that regard code from a humanities methodological point of view, 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. We have tried to sketch 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-reflexively turn criticism towards 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-reflexive. Methodology now increasingly means digital methodology, but that does not imply that critical self-reflexivity 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. We hope this paper may have contributed to the awareness that this is a critical task of digital humanities as well.

--JZ_20151029_1522

Notes

[1] All statements on the CollateX software pertain to the 2.0.0 version of the Python port available on the Pypi (https://pypi.python.org/pypi/collatex) Python library repository. The open source code is available under GPLv3 license at Github (https://github.com/interedition/collatex/tree/master/collatex-pythonport).

[2] For writerly text see https://en.wikipedia.org/wiki/The_Pleasure_of_the_Text.

[3] 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. Tool criticism might at some point very well be integrated into the approach we are suggesting here, but that is beyond the scope of this paper.

[3] Obviously visualization constitutes a transformation of the data that is being modeled too, and therefore it also constitutes an interpretation and argument about that data. Like tool criticism however, interface critique is out of scope here, although the *code* that drives visualization could be subject to code criticism within a code criticism framework.

References

Andrews, T. L. & Macé, C. (2013). Beyond the tree of texts: Building an empirical model of scribal variation through graph analysis of texts and stemmata. *Literary and Linguistic Computing*, 28(4), pp.504–521.

Bauer, J. (2011). Who You Calling Untheoretical? *Journal of Digital Humanities*, 1(1). Available at: http://journalofdigitalhumanities.org/1-1/who-you-calling-untheoretical-by-jean-bauer/.

Beaulieu, A., Dalen-Oskam, K. van and Zundert, J. van (2012). Between Tradition and Web 2.0: eLaborate as a Social Experiment in Humanities Scholarship. In T. Takševa, ed. *Social Software and the Evolution of User Expertise: Future Trends in Knowledge Creation and Dissemination*. IGI Global, pp. 112–129. Available at: 10.4018/978-1-4666-2178-7.ch007.

Berry, David (2014). *Critical Theory and the Digital*. Critical Theory and Contemporary Society. New York, London, New Delhi etc.: Bloomsbury Academic.

Coyne, Richard (1995). Designing Information Technology in the Postmodern Age: From Method to Metaphor. Leonardo Book Series. Cambridge: The MIT Press.

Galey, A. & Ruecker, S. (2010). How a prototype argues. *Literary and Linguistic Computing*, 25(4), pp.405–424.

Hacking, Ian (1981). "Do We See Through a Microscope?" *Pacific Philosophical Quarterly* 62 (4): 305–22.

Haentjens Dekker, Ronald, Dirk Van Hulle, Gregor Middell, Vincent Neyt, and Joris van Zundert (2014). "Computer Supported Collation of Modern Manuscripts:

CollateX and the Beckett Digital Manuscript Project." *Literary and Linguistic Computing*, March. doi:10.1093/llc/fqu007.

Huitfeldt, C. and Sperberg-McQueen, C. M. (2008). What is Transcription. *Literary and Linguistic Computing*, 23(3), pp.295–310.

Jones, Steven E. (2014). The Emergence of the Digital Humanities. Routledge.

Knuth, D. E. (1984). Literate Programming. *The Computer Journal*, 27(1), pp.97–111.

Lehman, M. M. and Ramil, J. F. (2000). Software evolution in the age of component-based software engineering. *Software, IEE Proceedings*, 147(6), pp.249–255.

Marino, M.C. (2006). Critical Code Studies. *Electronic Book Review*. Available at: http://www.electronicbookreview.com/thread/electropoetics/codology [Accessed January 16, 2015].

McGann, J. (2013). Philology in a New Key. *Critical Inquiry*, 39(2), pp.327–346.

McPherson, T. (2012). Why Are the Digital Humanities So White? or Thinking the Histories of Race and Computation. In M. K. Gold, ed. *Debates in the Digital Humanities*. Minneapolis: University of Minnesota Press, pp. 139–60. Available at: http://dhdebates.gc.cuny.edu/debates/text/11.

Montfort, N. et al. (2012). *10 PRINT CHR$(205.5+RND(1)); : GOTO 10*, Cambridge Mass.: MIT Press. Available at: http://10print.org/ [Accessed January 6, 2015].

Robinson, P. (1989). The Collation and Textual Criticism of Ice- landic Manuscripts (1): Collation. *Literary and Linguistic Computing*, 4(2), pp.99–105.

Roelli, P. (2015). Copy text. *Parvum Lexicon Stemmatologicum*. Available at: https://wiki.hiit.fi/display/stemmatology/Copy+text [Accessed October 29, 2015].

Unsworth, J. (2000). Scholarly Primitives: what methods do humanities researchers have in common, and how might our tools reflect this? In Symposium on "Humanities Computing: formal methods, experimental practice." London: King's College. Available at: http://people.brandeis.edu/~unsworth/Kings.5-00/primitives.html [Accessed June 27, 2014].

Wark, M. (2001). Essay: Codework. *American Book Review*, 22(6). Available at: http://amsterdam.nettime.org/Lists-Archives/nettime-l-0109/msg00197.html [Accessed October 28, 2015].

Zundert, J. J. van (2012). If you build it, will we come? Large scale digital infrastructures as a dead end for digital humanities. *Historical Social Research–Historische Sozialforschung*, 37(3), pp.165–186.

Zundert, J. J. van (2015). "Editor, Author, Engineer—Code & the Transformation of Authorship in Scholarly Editing." Submitted for publication.