WIENER digitale REVUE

Zeitschrift für Germanistik und Gegenwart

Simon Roloff

Form statt Funktion

Neue Tendenzen der Quellcodeanalyse

DOI: 10.25365/wdr-06-03-02

Simon Roloff

Form statt Funktion

Neue Tendenzen der Quellcodeanalyse

1. Eine neue Alphabetisierung

1 Quellcodes sind die wichtigste Textform unserer Zeit. Sie steuern Klimaanlagen, Fahrstühle und Flugleitsysteme, leiten Verwaltungsprozesse ein, geben medizinische Empfehlungen oder entscheiden über die Sichtbarkeit eines Posts in sozialen Netzwerken. In diesen und unzähligen weiteren Funktionen bleiben sie allerdings unsichtbar; allenfalls bei einer Fehlfunktion wird ihre Existenz bewusst, etwa wenn wir eine Bilddatei in unserem Mailprogramm öffnen und deren jpeg-Kodierung angezeigt bekommen. Weitaus gravierendere Folgen hatte im Sommer 2024 ein fehlerhaftes Software-Update der Firma Crowdstrike, das zu Systemausfällen bei Fluggesellschaften, Banken und Medien führte. Bei derartigen Störfällen erkennen wir plötzlich, dass maschinenlesbare Texte an wesentlicher Stelle einer „Kultur der Digitalität" stehen, in der „Algorithmizität" Informationen ordnet und Entscheidungsverfahren automatisiert, um auf diesem Weg in einer „Kultur der Digitalität" (Stalder 2016: 13) politische, ästhetische, ökonomische, kommunikative, rechtliche und pädagogische Prozesse zu steuern. „Codes sind das, was uns – vom Namen und von der Sache her – bestimmt, und was wir artikulieren müssen, und sei es nur, um nicht vollständig unter ihnen zu verschwinden". (Kittler 2008: 40)

2 In den deutschsprachigen Medienwissenschaften wurde diese Forderung Friedrich Kittlers lange Zeit überhört oder vielmehr überlagert von seiner früheren polemischen Einlassung „Es gibt keine Software" (Kittler 1993; vgl. Heilmann 2018). Quellcodes standen deshalb lange hinter der Hardware als Untersuchungsobjekt und bevorzugtem Zugang zum *medialen Apriori* (Kittler) zurück. Dabei kann ihre Analyse eigentlich anhand von Textkommentaren, also einer grundlegenden Praxis der Geistes- und Kulturwissenschaften geschehen. Der folgende Beitrag gibt einen Überblick über die bestehenden Ansätze, eine Expertise für Algorithmen in den traditionell mit Texten arbeitenden Disziplinen zu etablieren. Abschließend stellt er in Auseinandersetzung mit diesen Ansätzen einen Versuch der Weiterentwicklung des Verständnisses von Code als Form vor, der von 2023 bis 2024 im Rahmen des Forschungsprojekts *code/verstehen: Philologie und Theorie algorithmischer Quelltexte* an den Universitäten Basel und Lüneburg, gefördert durch die Volkswagen-Stiftung entwickelt wurde (vgl. VW-Stiftung 2022).

3 Wer Codes kommentieren will, muss sie zunächst einmal lesen können. Häufig stellt dabei schon der Zugang zu ihnen ein großes Hindernis dar – wie im Fall des Edge-Rank-Algorithmus, der darüber entscheidet, wie oft und welchen Personen ein Facebook-Post angezeigt wird. Kultur- und Geisteswissenschaftler:innen, ebenso wie Jurist:innen und Mediziner:innen, stehen aber zusätzlich vor der Herausforderung, dass die für ein Verständnis von Code erforderliche Lese- und Schreibkompetenz nicht zum Standard ihrer Ausbildung gehört. Selbst dort, wo es möglich ist, Open-Source-Programme zu analysieren oder die Freigabe von geschützter Software zu erwirken, bleibt ihnen deshalb ein informierter Zugang aufgrund mangelnder Kompetenzen oft verwehrt. Seit einiger Zeit wird daher eine *Code Literacy* der nicht technisch orientierten Fächer gefordert. Dies geschieht auch und vor allem unter dem Eindruck, dass computergestützte Methoden diese Fächer nachhaltig verändern: „Die Allgegenwart des Computers

bedeutet, dass das Programmieren aus der exklusiven Domäne der Informatik ausbricht und in Berufe wie Journalismus, Biologie, Design und – über Bibliotheksdatenbanken und digitale Geisteswissenschaften – sogar in die Literatur- und Geschichtswissenschaften eindringt." (Vee 2017: 15). Annette Vee begreift Codingkenntnisse angesichts dieser Entwicklung als eine „Plattformkompetenz" (ebd.), die Lesen und Schreiben im digitalen Zeitalter an die Seite gestellt und insofern anderen digitalen Kompetenzvermittlungsmodellen vorgelagert ist, welche eher auf eine gewisse Versiertheit im Umgang mit digitalen Benutzeroberflächen oder auf Kenntnisse im Umgang mit anderen Medien zielen. Gemeint sind erziehungswissenschaftliche Konzepte wie *Information Literacy*, *Data Literacy* und *AI Literacy*, die als *Soft Skills* im Kompetenzkanon längst Einzug gehalten haben (vgl. Hoechsmann 2012; Parry et al. 2017).

4 Der Begriff der *literacy*, also der Lese- und Schreibfähigkeit, ist allerdings mit schwerem historischem Gepäck beladen: Seit dem 18. Jahrhundert verbindet sich mit ihm nichts weniger als die Konstitution moderner Staatlichkeit auf dem Rücken von Alphabetisierungsprogrammen (vgl. Bosse 2015; Kittler 2003). Programmierfähigkeit und die Kenntnis von Programmiersprachen stehen demnach, solange man sie als Element einer grundlegenden *literacy* konzipiert, in einer Tradition der Pädagogik, die stets mit der moralischen Aufforderung an die Subjekte gearbeitet hat, zu aktiveren und wertvolleren Teilen der bürgerlichen Gesellschaft zu werden (vgl. Vee 2017: 72). Aktuell wird der Begriff stärker in Diskursen der Partizipation, der Chancengleichheit oder der Potenzialentfaltung benachteiligter Personengruppen positioniert (vgl. Mihailidis 2019) – auch hier zieht er aber, auf Codes und Coding angewandt, eine Verbindung zwischen dem neuen ABC des Digitalen und einer sonst nicht mehr selbstverständlichen produktiven Teilhabe an Kultur, Gesellschaft und Staat. Im universitären Bereich leitet sich daraus eine dringliche Handlungsaufforderung an die Ausbildungsstrukturen und die Fächerkultur der nicht technischen Disziplinen ab und hier stellt sich die Frage, ob man ihr ausschließlich mit defensiven oder dienstfertig nachholenden Maßnahmen begegnen will, oder ob man nicht vielmehr die eigenen Kompetenzen umgekehrt auf den neuen Gegenstand richtet und eine Gegenaufforderung an die traditionell mit Programmierkenntnissen hantierenden Fächer zur Neuperspektivierung ihrer Praxis ausspricht.

2. Quellcodekritik als Forschungsparadigma

5 Die *Critical Code Studies* (CCS), von Mark Marino schon im Jahr 2006 in Manifestform initiiert und seither in zahlreichen Studien weitergeführt, sind in mancher Hinsicht genau dieser Agenda verschrieben. Korrespondierend mit theoretischen Versuchen, Code als grundlegende sprachliche Kommunikationsform zu begreifen (vgl. Hayles 2010: 16) werden hier die kulturellen, sozialen und ästhetischen Dimensionen von Programmcode untersucht – in bewusster Abgrenzung zu rein technikgeschichtlichen Analysen, die etwa die historische Entwicklung von Programmiersprachen behandeln (vgl. Dowek/Lévy 2011; Salus 1998). Die CCS entdecken in Codes eine verborgene Bedeutungsebene, die implizit trotz ihrer rein instrumentellen Verwendung in der Kommunikation mit Maschinen entsteht: „Ich möchte vorschlagen, dass wir Code nicht mehr nur als Text im übertragenen Sinne behandeln, sondern ihn als Text analysieren und verstehen, als ein Zeichensystem mit eigener Rhetorik und semiotischer Kommunikation, das Bedeutung als Überschuss zu seinem funktionalen Gebrauch produziert." (Marino 2020a: 39) Die besondere Perspektive der CCS entsteht also aus einer Loslösung von Programmcodes aus ihrem Gebrauchskontext – unter der Annahme, dass sie niemals nur rein in Bezug auf technische Notwendigkeiten formuliert sind und also nie nur die Rolle einer neutralen Übersetzungsinstanz zwischen Mensch und Maschine spielen. Codes sind aus dieser Sicht vornehmlich ein Artefakt, in dem Ideologien, soziale Normen und kulturelle Formationen

ebenso eingeschrieben sind wie in Literatur, Kunst oder Musik. An sie stellt man hier deshalb Fragen einer bestimmten Form der literaturwissenschaftlichen Interpretation: Welche impliziten Werte und Überzeugungen sind ihnen eingeschrieben? Wie werden bestimmte Gruppen durch sie inkludiert oder exkludiert? Was sagen sie über die kulturellen Formationen aus, in denen sie entstanden sind? Die CCS folgen in ihrem Textverständnis also Disziplinen wie den *critical legal studies* oder *critical race studies*, deren Interpretationen sich auch auf ursprünglich in einem Gebrauchskontext entstandene Dokumente wie Urteilsbegründungen oder Quellen zum atlantischen Sklavenhandel richten können, um diskursive Überschüsse in ihnen zu analysieren (vgl. ebd.).

6 Zur Analyse von Codes können in den CCS auch andere Texte, ebenso wie materielle Artefakte herangezogen werden: so etwa die Dokumentation des Codes, oder die Kommentare der Entwickler:innen, aber auch ihre schriftliche Kommunikation untereinander. Relevant kann aber auch die Computerhardware werden, für die ein Programm entwickelt wurde, die Geschichte und der Entwicklungskontext der Sprachen, in denen es formuliert ist oder die Art und Weise ihrer Vermittlung in Handbüchern für Programmierer:innen. Ebenso spielt seine Herkunft aus Forschung, Unterhaltungsindustrie oder militärischer Nutzung für die Deutung eine Rolle. Andere Untersuchungen können die gerade von Computerspielen häufig hergestellten Referenzen auf Literatur, Kunst und Film in den Blick nehmen (vgl. ebd.: 44).

7 Der Sammelband *10 PRINT CHR$(205.5+RND(1)); : GOTO 10* demonstriert eine Möglichkeit, Programme im Rahmen eines CCS-Ansatzes zu interpretieren (für das Folgende: Montfort et al. 2012). Die titelgebende Codezeile stellt ein kurzes Programm dar, dessen Ausführung auf frühen Computern randomisierte Labyrinthgrafiken erzeugte. Die Beiträge des Bandes greifen unterschiedliche Aspekte des Codes auf, etwa um die kulturelle und wissenschaftliche Bedeutung von Labyrinthen in den 1980er Jahren zu untersuchen. Da das Programm eine Funktion für zufällige Generierungen enthält, wird auch der Zufall als Problem für Computeralgorithmen insgesamt thematisiert. Andere Beiträge zielen dagegen darauf ab, das Programm im Rahmen der Entwicklung der Programmiersprachen und seiner historischen Hardware zu kontextualisieren, also in den Ursprüngen der Programmiersprache BASIC und technischen Aspekten des Commodore C64, wie dem PETSCII-Zeichensatz, dem VIC-II-Videoprozessor und dem KERNAL-Betriebssystem. Auch ohne auf die jeweiligen Interpretationsbefunde an dieser Stelle näher einzugehen, tritt aus dieser Untersuchung einer äußerst kurzen Zeile Code *close reading* als wesentliches Verfahren der CCS hervor: Anhand einer einzigen Codezeile sind unterschiedliche metaphorische, epistemische oder technikhistorische Befunde möglich. Ebenso zeigt sich ein grundsätzlich historisierender Charakter des Forschungsansatzes, der in diesem Fall Besonderheiten des frühen Homecomputing, der Gaming-Kultur der 1980er Jahre, aber auch die wissensarchäologische Rahmung der Computerkultur insgesamt aufgreift.

8 Einige jüngere quellcodekritische Studien aus dem deutschsprachigen Raum knüpfen an die CCS an und entwickeln sie bereits in wesentlichen Elementen weiter. So wurde der Versuch unternommen, Programmtexte als Akteure in kulturtechnischen Operationsketten zu beschreiben und darauf aufbauend ein Vorschlag entwickelt, die traditionell von Programmierer:innen genutzte Praxis des Kommentierens von Code im Code auch zur kritischen Aufarbeitung der Programmlogik für eine nicht ausschließlich technisch interessierte Leserschaft zu nutzen (vgl. Krajewski 2024). Eine entscheidende Neuerung stellt auch der Versuch dar, die mit klassischen Leseverfahren nicht zu bewältigenden Textmassen vieler Programmcodes zunächst doch wieder durch maschinelle Lektüreverfahren, und zwar mit Methoden der Digital Humanities in einem *distant reading* zu bewältigen. Dies erscheint auch notwendig, wenn z. B. schon eine sehr frühe Version von Photoshop (1.0.1 aus dem Jahr 1990), deren Code heute öffentlich

zugänglich ist, aus ca. 100.000 Zeilen Text besteht. Es ist also sinnvoll, zunächst relevante Analysefelder einzugrenzen, z.B. indem man die Verwendung bestimmter Ausdrücke und Befehlsketten zunächst auf ihre Häufigkeit hin untersucht. Der Code wird so für eine semantisch orientierte Lektüre vorbereitet, die sich inhaltlich den Sprachen, Paradigmen, Kommentaren und Benennungen widmet (vgl. Heilmann 2024). Einige Untersuchungen befassen sich bereits mit dem Dilemma, dass die methodischen Innovationen des Forschungsfeldes vielleicht gerade zu einem Zeitpunkt entwickelt werden, in dem sich längst ein neues Programmierparadigma unter weitgehendem Verzicht auf von Menschen generierten Code abzeichnet. Bekanntlich basieren Künstliche Neuronale Netze der neuesten Generationen nicht mehr auf sequenziellen Befehlszeilen, deren logische Abfolge prinzipiell in einer zuvor von ihren Entwickler:innen codierten Form untersuchbar ist, sondern erzielen ihre Ergebnisse über tausende parallel ablaufende Prozesse, die in Wechselwirkung zueinanderstehen. Die so entstehende Unübersichtlichkeit macht es z.B. in großen Sprachmodellen unmöglich, die Entscheidung für bestimmte Generierungen in einer bestimmten Programmzeile zu verorten. Vor diesem Hintergrund wird das theoretische Verständnis von KI-Architekturen in Bezug auf Sprachverarbeitung und Semantik aktuell intensiv diskutiert (vgl. Kirschenbaum 2024, Bajohr 2024).

3. Code als Form

9 Während seit einiger Zeit also kein Zweifel mehr besteht, dass Algorithmen in Textform nicht ausschließlich für Maschinen geschrieben, sondern auch von Menschen gelesen und kritisch analysiert werden können, so zeichnen sich mit Blick auf die bisherigen Ansätze doch auch einige methodische Probleme des Forschungsfeldes ab. An erster Stelle steht hier die schon erwähnte Zugänglichkeit des Textkorpus. Es liegt in der Natur der Sache, dass Code-Forscher:innen nur auf Open-Source-Programme zugreifen können, was wiederum große privatwirtschaftliche Anwendungen wie z.B. Social-Media-Algorithmen oder Suchmaschinen-Logiken ausschließt – trotz ihrer offensichtlichen Relevanz für Untersuchungen zur digitalen Kultur der Gegenwart. In Platform Studies, Software Studies und Science und Technology Studies, deren methodischer Zugang zu digitalen Artefakten auch nicht auf Text beschränkt ist, können solche Anwendungen dagegen diskutiert werden. Viele Forschungsbeiträge aus den CCS thematisieren aufgrund dieser Zugangsproblematik historische Codes, die im Fall kommerzieller Anwendungen nachträglich freigegeben wurden (wie im Fall der bereits genannten frühen Photoshop-Version). Das muss per se kein Makel sein, setzt den möglichen Feldern einer eng als Textanalyse gefassten Quellcodekritik aber doch Grenzen.

10 Das zweite Problemfeld geht direkt aus der Orientierung der CCS an der *critique* und der aus ihr abgeleiteten Konzentration auf die natürlichsprachliche Metaphorik von Programmsprachen hervor. Es ist verführerisch – und kann produktiv sein – Quellcodes als ideologischen Text zu dechiffrieren, in dessen Variablenbenennungen und Kommentaren verborgene Bedeutungsüberschüsse angelegt sind. Allerdings hat die von den CCS oft bemühte Analogie zwischen Source Codes und literarischen Texten eine Grenze: Je nach verwendeter Programmiersprache sind natürlichsprachliche Begriffe im Programmtext nur in sehr geringfügigem Umfang vorhanden. Und selbst wo dies in neueren, auf Benutzerfreundlichkeit ausgelegten Sprachen der Fall ist, sind diese Begriffe zu einem Großteil durch die Konventionen der verwendeten Programmiersprachen vorgegeben. Natürlich können auch diese Konventionen in einem Kommentar zum Thema werden, aber es ist kein Zufall, dass *close reading* als Methode der Analyse eines Codes besonders fruchtbar dann zu sein scheint, wenn er – wie im Fall des von Marino analysierten *Transborder Immigrant*

Tools – bereits im Hinblick auf seine Lesbarkeit als Text mit Bedeutungsüberschuss geschrieben wurde: Als konzeptuelle Aussage eines digitalen Kunstprojekts ist seine Terminologie bewusst darauf ausgerichtet, einen poetischen Referenzrahmen für den Algorithmus zu eröffnen (vgl. Marino 2020b).

11 Während es also zweifellos Codes gibt, deren literarische Mittel in einem *close reading* analysiert werden können, stellt sich doch die Frage nach dem heuristischen Vorgehen, wenn dies nicht der Fall sein sollte. Schließt man die entsprechenden Codes, obwohl sie für eine Forschungsfrage relevant sein können, von einer kritischen Analyse aus? Sinnvoller erscheint es, den Instrumentenkasten der Interpretation zu erweitern, wie dies in einigen Beiträgen des Sammelband code/verstehen unternommen wird (vgl. Bajohr/Roloff 2025). Und hier könnte eine Perspektivverschiebung von den sprachlichen Mitteln eines Codes hin zu seiner Form eine entscheidende Rolle spielen. Die Betonung der Form eines Codes in der Analyse ist auch im Fall sequenziell arbeitender Algorithmen eine naheliegende Grundsatzentscheidung, denn es gibt wohl kaum stärker formalisierte Texte als sie. Nur eine rigide Syntax und konventionalisierte Wortwahl sowie die Einhaltung verschiedenster Programmierkonventionen lassen sie ihre Funktionen erfüllen – jeder formale Fehler führt zum Abbruch der Ausführung des Programms. Weil höhere Programmiersprachen beständig die Waage zwischen natürlicher Sprache und einer für den Compiler verständlichen Form halten, erscheint uns das Verhältnis zwischen ihnen und den Vorgängen, die sie beschreiben, auch häufig so problematisch – die Geschichte der Computerprogrammierung ist voller Beispiele für fehlgehende Reduktionen komplexer Vorgänge auf wenige, maschinell verarbeitbare Algorithmen. Quellcodes stellen notwendig „dünne Regeln“ auf, wie Lorraine Daston dies nennt, sie gehen „von einer vorhersehbaren, stabilen Welt aus, in der alle Möglichkeiten vorhersehbar sind und laden nicht zur Ausübung von Ermessensspielraum ein.“ (Daston 2022: 3) Trotzdem werden die kontinuierlichen und dynamischen Prozesse, die wir als Menschen wahrnehmen und in denen wir existieren, natürlich immer weiter in eine notwendig diskontinuierliche und statische Version ihrer selbst für die Verarbeitung in Computern übertragen. Und an diesem Punkt lassen sie sich zur Form literarischer Texte in Beziehung setzen, die immer da im Zentrum textgebundener Darstellung entsteht, wo „Stabiles auf Dynamisches, Gesteuertes auf Emergentes trifft“ (Erdbeer 2022: 11). Die Formanalyse eines Codes bezieht demnach all jene Elemente ein, in denen seine äußere Erscheinung als Text der Welt ein eigenes Gefüge gibt. Das bedeutet, dass wir z.B. die Grammatik und Syntax von Programmiersprachen, die Programmierkonventionen, sowie die Verwendung von Datenstrukturen in einer Software als Ordnungs-, Strukturierungs-, Klassifikations- und Wertungspraktiken beschreiben können.

12 Wenn Code sich also aufgrund seiner logischen Reduktion ontologischer Gegebenheiten und ihrer Neuverknüpfung in einer künstlichen Konstruktion besonders für eine Formanalyse anbietet, so ist diese Form nicht einfach durch „den Computer“ vorgegeben. Sie ist vielmehr das Ergebnis einer Spannung zwischen seiner binären Schaltung, der algorithmischen Logik des Programms, sowie, drittens, seiner konkreten Ausführung im Stil der jeweiligen Programmierer:innen, die auf unterschiedliche Weise historisch wandelbaren, normativen Vorgaben wie Effizienz, Eleganz oder Nachvollziehbarkeit folgen – ähnlich den „epistemischen Tugenden“ der Wissenschaft, die Verfahren und Vorgehen ihrer Erkenntnis bestimmen (Daston/Galison 2007: 28). Die Form eines Codes ist also etwas, das aus der Technik der ihn ausführenden Maschine, den Praktiken des Programmierens, den Regeln seiner Sprache und aus den ihn umgebenden epistemischen Ordnungen entsteht. Sie ist daher nicht durch bloße Aufmerksamkeit für die Textoberfläche, ihre Rhythmen, Muster, Wiederholungen und deren Beschreibung analysierbar (vgl. Kramnick/Nersessian 2024: 653f.). Sie ist auch nicht zur Gänze jenseits ihres historischen Entstehungskontextes zu bestimmen, wie dies einige revisionistische Positionen der neo-formalistischen

Literaturwissenschaft voraussetzen (vgl. Felski 2008, 2015; MacPherson 2015). Mit dem reduktionistischen Formalismus von Carolie Levine muss dagegen die horizontale Verbindung von Code zu anderen Formen – sozialen, juristischen, epistemischen – auf dem Feld einer gemeinsamen Geschichte beschrieben werden. Formen sind bei Levine nie nur ästhetische Phänomene oder eine alleinige Domäne des Textes. Sobald man sie in ihren Affordanzen erfasst, also in ihrer Fähigkeit, Ordnung, Struktur und Gestalt in einem Sujet herzustellen, können sie auch in ihrer Verknüpfung mit anderen, sie umgebenden Formen beschrieben werden. Sie spiegeln dabei „Konfigurationen, Ordnungsprinzipien und Mustern von Wiederholung und Differenz" jenseits des Codes in einer Gesellschaft und Kultur, unterstützen oder stabilisieren sie dabei, oder stellen sie infrage (Levine 2015: 3).

4. Die Form des *Forensic Statistical Tool*

13 Das Potenzial einer formalen Analyse von Quellcodes lässt sich exemplarisch am *Forensic Statistical Tool* (FST) aufzeigen. Diese Software galt zum Zeitpunkt ihrer Entwicklung als Pionierarbeit auf dem Gebiet des statistischen DNA-Abgleichs, einem relativ neuen forensischen Verfahren. Wenn DNA-Proben von einem Tatort das Erbgut mehrerer Personen enthalten, ist der klassische DNA-Abgleich, der aufgrund der Einzigartigkeit der DNA-Struktur einer Person als besonders sicheres Beweismittel gilt, nicht mehr durchführbar. Er benötigt weitgehend unverfälschte Proben von einer eindeutig zu identifizierenden Person. Hier kommen nun Anwendungen wie das FST ins Spiel, die auch komplex zusammengesetzte Spuren analysieren können, in denen DNA von mehreren Personen enthalten ist. Sie geben allerdings nur die Wahrscheinlichkeit an, mit der die DNA einer verdächtigen Person im Vergleich zu einer zufälligen Person der gleichen ethnischen Gruppe in der Probe enthalten ist oder umgekehrt die Wahrscheinlichkeit, mit der dies eben nicht der Fall ist. Der Algorithmus des FST verwendet dafür ein statistisches Vorhersageverfahren, das zunächst alle möglichen Rekombinationen für die in der Probe gefundenen Genausprägungen (Allele) generiert, um diese dann mit einer Populationsdatenbank und dem DNA-Profilen eines Tatverdächtigen zu vergleichen. Daraus leitet das Programm einen numerischen Wert ab, der die gefundenen DNA-Kombinationen mit der Vergleichsprobe in Beziehung setzt (vgl. Watters/Coyle 2016). Die Software wurde vom Labor des Office of the Chief Medical Examiner (OCME) in New York City entwickelt und von der New Yorker Polizei wahrscheinlich bei tausenden Ermittlungen eingesetzt (vgl. Kirchner 2017).

14 Digitale Anwendungen für polizeiliche oder gerichtliche Zwecke sind trotz ihrer sensiblen Natur und den weitreichenden Konsequenzen ihrer Berechnungen in ihrer Programmlogik in der Regel nicht öffentlich zugänglich, da es zumindest in den USA üblich ist, solche Anwendungen in privatwirtschaftlichen Strukturen zu entwickeln. Das bedeutet, dass ihr Code als Geschäftsgeheimnis geschützt ist. Im Fall einer DNA-Abgleich-Software kennen Polizei, Staatsanwaltschaft, Gericht, Verteidigung und sogar das beteiligte DNA-Labor daher in der Regel nur die Fehlerquoten ihres Tools, nicht aber den Berechnungsalgorithmus selbst und können daher auch nicht überprüfen, welche Kriterien seinen Lösungen zu Grunde liegen (vgl. Wexler 2015). Auch der Code des FST, obwohl von einer öffentlichen Einrichtung entwickelt, war zunächst nicht einsehbar. Trotzdem konnte er in Gerichtsverfahren eingesetzt und im Extremfall zu Verurteilungen führen, bei denen die Angeklagten ausschließlich durch die Software belastet wurden. So etwa im Strafverfahren gegen die Mitglieder einer jüdischen Bürgerwehr wegen einer Prügelattacke in New York im Jahr 2013: Einer der Angeklagten musste hier zunächst eine hohe Haftstrafe antreten, obwohl er jegliche Beteiligung an der Tat bestritt und keine weiteren Indizien oder Beweise gegen ihn vorlagen, was einige mediale Aufmerksamkeit erregte und sogar in einer True-Crime-Serie aufgegriffen

wurde (vgl. Loudenberg 2019). Das FST hatte eine 133-fache Wahrscheinlichkeit errechnet, dass eine DNA-Spur am Schuh des Opfers mit der des Beschuldigten identisch war (vgl. Kirchner 2017). Da jedoch immer wieder Zweifel an der Zuverlässigkeit des Algorithmus geäußert wurden – prominent unter anderem von einem ehemaligen technischen Leiter des DNA-Labors des OCME – erreichte die Organisation ProPublica im Jahr 2017 die Veröffentlichung des FST-Codes zusammen mit begleitenden Materialien wie dem Benutzerhandbuch und einer Executive Summary, die seitdem auf dem Programmrepositorium GitHub einsehbar sind (vgl. Kirchner et al. 2017). Seitdem besteht die seltene Gelegenheit, ein neuartiges forensisches Programm mit Relevanz für Fragen der gegenwärtigen Polizeiarbeit und ihrer Beziehung zur Statistik als Regierungstechnik über seine Benutzeroberflächen, das Begleitmaterial mit Hinweisen zur praktischen Anwendung der Software und den Code zur Berechnung der DNA-Wahrscheinlichkeitswerte eingehend zu analysieren.

15 Der FST liegt auf GitHub in einer Ordnerstruktur vor, die zum größten Teil Dateien in der Programmiersprache C# und zu einem kleineren Teil in Java enthält. In C#-Projekten werden Ordnerstrukturen verwendet, um das Programm in sogenannten Namespaces zu organisieren. Namespaces stellen eine Ordnungsstruktur innerhalb des Codes dar, die zusammengehörige Klassen und andere Funktionen zusammenfasst, um sie leichter verwalten zu können. Für die folgende Analyse ist der Namespace FST.Commons relevant, da hier die wesentliche Berechnungslogik des Programms in der Datei **comparison.cs** enthalten ist. Weitere Dateien in FST.Commons regeln den Zugriff des Programms auf die Vergleichsdatenbanken und die Ausgabe der Ergebnisse. Der Code in **comparison.cs** initiiert im Wesentlichen einen *Likelihood-Ratio-Test* (LR-Test), bei dem die Wahrscheinlichkeit zweier Hypothesen bewertet wird: Einerseits die Wahrscheinlichkeit, dass die DNA vom Verdächtigen stammt und andererseits die Wahrscheinlichkeit, dass die DNA von einer Vergleichsperson oder einer zufälligen Person aus derselben ethnischen Population stammt. Das Programm beginnt in der Klasse **Comparison** mit der Initialisierung der DNA-Profile aus einem Beweisstück, einer Vergleichsprobe sowie bekannten, nach Ethnien sortierten Profilstrukturen in einer Datenbank, von der später noch die Rede sein wird (die Nummerierung der Zeilen des Codes entspricht im Folgenden seiner Dokumentation unter Kirchner et al. 2017).

Abbildung 1. Code-Ausschnitt aus dem *Forensic Statistical Tool* (FST).

```
142  public Comparison
143  {
144      // Initialisierung der Dropout- und Drop-in-Raten sowie der
             Allelfrequenzen
145  public Dictionary<string, Dictionary<string, double>>
           perRaceLocusDenominatorCache = null;
146        public Dictionary<string, Dictionary<string, Dictionary<string,
           float>>> perRaceLocusFrequencyCache = null;
147        public DataTable raceTable = null;
148
149      public Comparison()
150      {
151          // Lade die ethnischen Daten und Dropout-Raten
152          myDb = new Database();
153                raceTable = myDb.getAllEthnics();  // Tabelle mit
                Populationsdaten
```

Abbildung 2. Code-Ausschnitt aus dem *Forensic Statistical Tool* (FST).

```
154                 numeratorDropOutRateTable =
                    ReadDropoutRate(dropoutOption, numPersonsInvolved,
                    strDeducible, labKitID);
155                 denominatorDropOutRateTable =
                    ReadDropoutRate(dropoutOption, denPersonsInvolved,
                    strDeducible, labKitID);
156         }
157
158         // Methode zur Berechnung der Vergleichsdaten
159         public Dictionary<string, float> DoCompare(DataTable dtFrequencies
            = null, ...)
160         {
161             DNAComparisonResult = new Dictionary<string, float>();
```

16 In dieser Passage des Codes findet noch keine Berechnung statt, aber man kann bereits die Verwendung von Datenstrukturen als wesentliches Instrument der Programmierung in **comparison.cs** erkennen: Die Verwendung von sogenannten *Dictionaries* fällt durch das wiederholte Auftreten des Begriffs im Code schon bei oberflächlicher Durchsicht ins Auge (Zeile 145, 146, 159). Dictionaries speichern Daten in sogenannten Schlüssel-Wert-Paaren und eignen sich besonders, wenn Daten schnell und ressourcensparend abrufbar sein sollen. In den Zeilen 154 und 155 werden indessen zwei Variablen definiert, die jeweils auf die Methode **ReadDropoutRate()** (Zeile 1396–1487) im Programm verweisen. Von dieser Methode wird das Datenformat *DataTable* zurückgegeben (Zeile 1484), das ähnlich einer Tabelle in einer Datenbank aus Zeilen und Spalten besteht, in dem also Daten mehrdimensional zueinander in Beziehung gesetzt werden können. Sowohl Dictionaries als auch DataTables können in C# verwendet werden, um Werte einander leicht und übersichtlich für den Zugriff durch den Algorithmus zuzuordnen. Sie erzeugen Adressierbarkeit, Sichtbarkeit und Nachvollziehbarkeit für die im Programm verwendeten Daten und stehen durch ihre Positionierung innerhalb der Definition der übergeordneten Klasse **comparison** an entscheidender Stelle im Vergleichsvorgang des FST.

17 Allein durch eine Wiederholung im Code (wobei die Definition von Methoden in Programmiersprachen wie Java, C# oder Python an sich schon eine Wiederholungsstruktur darstellt, da der Code innerhalb einer Klasse immer wieder aufgerufen werden kann) lässt sich in ihm also die Verwendung von Datenstrukturen als wesentliches Element seines Verfahrens erkennen. Und hier verweist die Form des FST auf die Anfänge der statistischen forensischen Praxis zu Beginn des Zwanzigsten Jahrhunderts. Insbesondere die Arbeit von Pearson (1934) über die Echtheit der Schädel von Henry Stewart und Oliver Cromwell kommt hier als Vorläufer in Betracht (vgl. Slice 2018). Bei diesen Forschungen spielte der Computer natürlich noch keine Rolle, sondern es wurden menschliche Rechenkräfte zur Bewältigung des enormen Rechenaufwands eingesetzt. Sie waren auch und vor allem mit der Erstellung und Auswertung des Generalinstruments aller statistischen Berechnungen beschäftigt: der Datentabelle, deren Verknüpfung von Zahlenwerten in Herstellung ihrer Vergleichbarkeit auf einer grafischen Oberfläche entscheidend für die Entstehung der Statistik als wesentlichem Instrument der Gesellschaftswissenschaften war (vgl. Stigler 2002: 48; Hacking 2010: 105–141). Das FST überträgt in Durchführung seiner zentralen Rechenoperation diese grundlegende Kulturtechnik des statistischen Denkens in den virtuellen Raum zeitgenössischer forensischer Polizeiarbeit.

18 Ein weiterer historischer Bezug der Form des Programms auf seine Anfänge in der frühen Statistik entsteht durch seine Verwendung der Korrelation als Berechnungsmethode. Im forensischen DNA-Vergleich des FST wird dieses ursprünglich von Francis Galton entwickelte Verfahren beispielsweise in der Methode **DoCompare()** (Zeile 336–547) verwendet, in der die Allelfrequenzen verschiedener ethnischer Populationen korreliert werden, um die Wahrscheinlichkeit zu bestimmen, mit der ein bestimmtes Allel (oder eine bestimmte Allelkombination) der entnommenen DNA-Probe in ihnen jeweils vorkommt. Dies wiederum ist für korrekte Berechnungen im FST von entscheidender Bedeutung, da Allelkombinationen in verschiedenen Populationen mit unterschiedlicher Häufigkeit vorhanden sind. Kommt also eine am Tatort vorgefundene Genkombination in einer Population kaum vor, wird aber bei einem Tatverdächtigen mit einer ethnischen Herkunft festgestellt, in der sie häufiger anzutreffen ist, so kann dies fälschlicherweise zur Berechnung eines hohen Wahrscheinlichkeitswerts seiner Anwesenheit am Tatort führen. Eine nach Ethnien differenzierende Behandlung des genetischen Materials ist also eine sinnvolle Funktion des Programms – zugleich zeigt sich hier in der Form die unauflösliche Verbindung der computergestützten Datenverwaltung und ihrer Techniken mit der rassistischen Wissenschaft der Eugenik, die bekanntlich von eben jenem bereits erwähnten Francis Galton maßgeblich begründet wurde und mit statistischen Methoden ermittelte genetische Unterschiede zwischen verschiedenen Ethnien zum Ansatzpunkt genetischer Züchtungsphantasien werden ließ: „Die Korrelation, die auf eugenischen Rekonstruktionen der Vergangenheit beruht [...], trägt den Keim der Manipulation, der Segregation und der Falschdarstellung in sich“ (Chun 2021: 58). Korrelationsberechnungen können überdies auch bei fehlerhaften Voraussetzungen in ihren Daten eine hohe wissenschaftliche Genauigkeit suggerieren. Dies war beim FST der Fall, dessen Algorithmus, wie die durch ProPublica beauftragte gutachterliche Überprüfung feststellte, ähnliche Datenstrukturen von Familienmitgliedern in der Berechnung nicht berücksichtigte. Wo die eugenischen Wissenschaft also Menschen in falschen Kategorien zusammenfasste, die auf oberflächlicher statistischer „Ähnlichkeit“ basierten (ebd.), bezieht das FST bestimmte Besonderheiten von Bevölkerungsgruppen nicht in seine Berechnung ein. Der Verurteilte in der erwähnten Prügelattacke entstammte etwa dem Milieu der orthodox lebenden chassidischen Juden in Williamsburg. Die Software berücksichtigte in ihrem Korrelationsalgorithmus aber diesen Fall einer kleinen Bevölkerungsgruppe nicht, in der seit jeher häufig untereinander geheiratet wurde und deren Mitglieder daher vermutlich mehr DNA miteinander teilen als andere Ethnien (vgl. Kirchner 2017).

Abbildung 3. Tabelle 4 der Bedienungsanleitung des FST (Anonym 2016: 24).

If the likelihood ratio is…	**Then the evidence provides…**
Less than 0.001	Very strong support for H_d over H_p
0.001 to 0.01	Strong support for H_d over H_p
0.01 to 0.1	Moderate support for H_d over H_p
0.1 to 1.0	Limited support for H_d over H_p
1 to 10	Limited support for H_p over H_d
10 to 100	Moderate support for H_p over H_d
100 to 1000	Strong support for H_p over H_d
Greater than 1000	Very strong support for H_p over H_d

19 Durch eine weitere Tabelle wurden die Ergebnisse des FST für die Weiterverarbeitung in Gerichtsverfahren vorbereitet. Im Benutzerhandbuch wird der numerische Wert der LR-Berechnung in eine qualitative Bewertung für ein wissenschaftliches Gutachten übersetzt, die FST ausgegebenen Quantität Werte also qualitativ beschrieben und die Bedienungsanleitung des FST legt an dieser Stelle großen Wert darauf, dass ihr „exact wording" verwendet wird (Ebd. 23). Ein LR von 1 bis 10 stützt in „begrenztem" Umfang die Annahme, dass die Verdächtigen-DNA in der Tatort-DNA vorhanden ist, von 10 bis 100 gilt diese These als „mäßig" belegt, ab einem Wert von 100 bis 1000 findet sich ein „starker" und von 1000 an ein „sehr starker" Hinweis darauf. Negative Werte werden analog in einer Skala für die Unterstützung der Ausschlusshypothese dargestellt: Von -1 für eine „begrenzte" Unterstützung bis -1000 für eine „sehr starke" Unterstützung, falls also das Genmaterial einer Person mit sehr hoher Wahrscheinlichkeit nicht in der DNA-Mischung vertreten ist. Zahlenwerte als Output des FST finden sich hier in eine Skala nüchterner, aber durchaus suggestiver Adjektive übersetzt, ohne dass die leitenden Annahmen für die qualitative Bewertung im Handbuch begründet werden und ohne dass potenzielle Gutachter:innen oder das Gericht Einblick in die Berechnungsweise des FST erhalten.

20 In der formalen Analyse eines Codes kann also die Gegenwart und Vergangenheit eines Algorithmus als normatives Instrument hervortreten – in diesem Fall das Erbe der rassistischen Wissenschaft in der Polizeisoftware der Gegenwart. Sie zeigt auch, wie die Form einer Software sich mit anderen normativen Ordnungen verbindet – in diesem Fall als eine Operationskette zwischen Tabellen und mathematischen Berechnungen, die das unsichere Feld der Wahrscheinlichkeiten aus der statistischen Untersuchung von DNA-Allelen in die qualifizierenden Festlegungen juristischer Gutachten übersetzen. Ein Quellcode überträgt also die Analyse eines DNA-Codes in die Sprache des *codex juridis*. Das FST ist hier mehr als ein komplexes Rechenwerkzeug, das die wissenschaftlichen Operationen forensischer DNA-Abgleiche mit komplexen Proben automatisiert. Es ist eine *Formoperation*, die epistemische Praktiken der Lebenswissenschaften in den institutionellen Text eines Gutachtens überträgt und auf diese Weise in die normative Ordnung juristischer Verfahren einpasst. Code ist an dieser Stelle das Instrument eines *vitam instituere* (Pierre Legendre), mithin einer Regierungstechnik, die die statistische Gliederung von Bevölkerungen (mit ihrer mal mehr mal weniger unschuldigen Vergangenheit) mit der Identifikationspraxis polizeilicher Ermittlungen verknüpft (vgl. Rouvroy 2013: 144).

Literaturverzeichnis

Anonym (2016): Forensic Statistical Tool (FST), Dokument 103:19 im Fall 1:15-cr-00565-VEC, S. 24. Abgerufen von www.documentcloud.org/documents/4113878-1-19-17-Exhibit-S-FST-Protocols-25-Pp.html, Zugriff am 08.10.2024.

Bajohr, Hannes (2024): Dumme Bedeutung. Künstliche Intelligenz und artifizielle Semantik, in: Krajewski, Markus/Bajohr, Hannes (Hg.): Quellcodekritik. Zur Philologie von Algorithmen. Berlin: August Verlag, S. 197–216.

Bajohr, Hannes/Roloff, Simon (2025) (Hg.): code/verstehen, Theorie und Philologie algorithmischer Quelltexte, Zürich: intercom (in Vorb.).

Bosse, Heinrich (2015): Die Schüler müssen selbst schreiben lernen: Über die Einführung der Schiefertafel, in: Zanetti, Sandro (Hg.): Schreiben als Kulturtechnik: Grundlagentexte. Berlin: Suhrkamp, S. 67–111.

Chun, Wendy (2021): Discriminating Data: Correlation, Neighborhoods, and the New Politics of Recognition. Cambridge, MA: The MIT Press.

Daston, Lorraine J. (2022): Rules: A Short History of What We Live By. Princeton: Princeton University Press.

Daston, Lorraine J./Galison, Peter (2007): Objectivity. New York: Zone Books.

Dowek, Gilles/Lévy, Jean-Jacques (2011): Introduction to the Theory of Programming Languages. London: Springer.

Erdbeer, Robert Matthias et al. (2022): Literarische Form, in: Ders. et al. (Hg.): Grundthemen der Literaturwissenschaft: Form. Berlin: De Gruyter, S. 3–69.
Felski, Rita (2008): Uses of Literature. Malden, MA: Blackwell Publishing.
Felski, Rita (2015): The Limits of Critique. Chicago: University of Chicago Press.
Hacking, Ian (2010): The taming of chance, Cambridge: Cambridge University Press.
Hayles, Katherine (2010): My Mother Was a Computer: Digital Subjects and Literary Texts. Chicago: University of Chicago Press.
Heilmann, Till A. (2018): Es gibt keine Software. Noch immer nicht oder nicht mehr, in: Ruf, Oliver (Hg): Smartphone-Ästhetik: Zur Philosophie und Gestaltung mobiler Medien. Bielefeld: transcript Verlag, S. 159–178.
Heilmann, Till A. (2024): Wie liest man 100.000 Zeilen Code? in: Krajewski, Markus/Bajohr, Hannes (Hg.): Quellcodekritik. Zur Philologie von Algorithmen. Berlin: August Verlag, S. 87–127.
Hoechsmann, Michael (2012): Media Literacies - A Critical Introduction. Chichester: Wiley-Blackwell.
Kirchner, Lauren (2017): Traces of Crime: How New York's DNA Techniques Became Tainted. The New York Times. Abgerufen von www.nytimes.com/2017/09/04/nyregion/dna-analysis-evidence-new-york-disputed-techniques.html, Zugriff am 16.09.2024.
Kirchner, Lauren/Klein, Scott/Tigas, Mike (2017): NYC DNA Software. Abgerufen von https://github.com/propublica/nyc-dna-software, Zugriff am 07.10.2024.
Kirschenbaum, Matthew G. (2024): Spec Acts: Formales Lesen in rekurrenten neuronalen Netzen, in: Bajohr, Hannes/Krajewski, Markus (Hg.): Quellcodekritik: Zur Philologie von Algorithmen. Berlin: August Verlag, S. 144–194. DOI: https://doi.org/10.52438/avaa1004.
Kittler, Friedrich A. (1993): Es gibt keine Software, in: Ders.: Draculas Vermächtnis: Technische Schriften, Leipzig: Reclam, S. 225–242.
Kittler, Friedrich A. (2003): Aufschreibesysteme 1800–1900. 4., vollst. überarb. Neuaufl. München: Fink.
Kittler, Friedrich A. (2008): Code (Or How to Write Something Differently), in: Matthew Fuller (Hg.): Software Studies: A Lexicon. Cambridge, MA: MIT Press, S. 40–46.
Krajewski, Markus (2024): Kulturtechnik Programmieren. Quellcode kritisieren. Drei Beispielszenarien, in: Ders./Bajohr, Hannes (Hg.): Quellcodekritik. Zur Philologie von Algorithmen. Berlin: August Verlag, S. 63–86. DOI: https://doi.org/10.52438/avaa1004.
Kramnick, Jonathan/Nersessian, Anahid (2024): Form and Explanation, in: Critical Inquiry 43/3, S. 650–669.
Levine, Caroline (2015): Forms: Whole, Rhythm, Hierarchy, Network. Princeton: Princeton University Press.
Loudenberg, Kelly (2019): Touch DNA. Exhibit A, Staffel 1, Episode 4, Netflix.
MacPherson, Sandra (2015): A Little Formalism, in: English Literary History 82/2, S. 385–405.
Marino, Mark C. (2020a): Critical Code Studies: A Manifesto, in: Ders.: Critical Code Studies. Cambridge, MA: The MIT Press, S. 37–54.
Marino, Mark C. (2020b): The Transborder Immigrant Tool, in: Ders.: Critical Code Studies. Cambridge, MA: The MIT Press, S. 55–104.
Mihailidis, Paul (2019): Civic Media Literacies, in: The International Encyclopedia of Media Literacy. New York: John Wiley & Sons, S. 1–10.
Montfort, Nick et al. (2012): 10 PRINT CHR$(205.5+RND(1));: GOTO 10. The MIT Press.
Parry, Becky/Burnett, Cathy/Merchant, Guy (Hg.) (2017): Literacy, Media, Technology: Past, Present and Future. London: Bloomsbury Academic.
Pearson, Karl (1934): The Wilkinson Head of Oliver Cromwell and its relationship to busts, masks and painted portraits. Biometrika 26, S. 1–116.
Rouvroy, Anette (2013): The End(s) of Critique: Data Behaviourism vs. Due Process, in: Hildebrandt, Mireille/Vries, Katja de (Hg.): Privacy, Due Process and the Computational Turn: The Philosophy of Law Meets the Philosophy of Technology. New York: Routledge, S. 143–167.
Salus, Peter H. (Hg.) (1998): Handbook of programming languages. 1. Aufl. Indianapolis/IN: Macmillan Technical Pub.
Slice, Dennis (2018): The Development and Use of Computational Tools in Forensic Science, in: Human Biology 90/3, S. 231–235.
Stalder, Felix (2016): Kultur der Digitalität, Berlin 2016.
Stigler, Stephen M. (2002): Statistics at the Table: The History of Statistical Concepts and Methods, Cambridge, MA: Harvard University Press.
Vee, Anette (2017): Coding Literacy: How Computer Programming is Changing Writing. Cambridge, MA: The MIT Press.

VW-Stiftung (2022): code/verstehen – Theorie und Philologie algorithmischer Quelltexte, Initiative Aufbruch – neue Räume für die Geistes- und Kulturwissenschaften. Abgerufen von https://portal.volkswagenstiftung.de/search/projectPDF.do?projectId=10346, Zugriff am 10.11.2024.

Watters, Kyle B./Coyle, Heather Miller (2016): Forensic Statistical Tool (FST): A Probabilistic Genotyping Software Program for Human Identification, in: Forensic Science Publications 56/2, S. 183–195. Abgerufen von https://digitalcommons.newhaven.edu/forensicscience-facpubs/50, Zugriff am 07.10.2024.

Wexler, Rebecca (2015): Convicted by Code, in: slate.com. Abgerufen von https://perma.cc/ZBD4-BY7G, Zugriff am 05.09.2024.

Zusammenfassung

Quellcodes bleiben in der Regel unsichtbar. Dabei kann ihre Analyse eigentlich anhand von Textkommentaren, also einer grundlegenden Praxis der Geistes- und Kulturwissenschaften geschehen. Der folgende Beitrag gibt einen Überblick über die bestehenden Ansätze, eine Expertise für Algorithmen in den traditionell mit Texten arbeitenden Disziplinen zu etablieren. Abschließend stellt er in Auseinandersetzung mit diesen Ansätzen einen Versuch der Weiterentwicklung des Verständnisses von Code als Form vor, der von 2023 bis 2024 im Rahmen des Forschungsprojekts *code/verstehen: Philologie und Theorie algorithmischer Quelltexte* an den Universitäten Basel und Lüneburg, gefördert durch die Volkswagen-Stiftung entwickelt wurde (VW-Stiftung 2022).

Schlagwörter: Quellcode, Algorithmen, Code als Form

Abstract

Source codes usually remain invisible. Yet they can actually be analyzed on the basis of text annotations, a fundamental practice of the humanities and cultural studies. The following article provides an overview of existing approaches to establishing expertise in algorithms in disciplines that traditionally work with texts. It concludes with a discussion of these approaches and presents an attempt to further develop the understanding of code as a form, which was carried out from 2023 to 2024 as part of the research project *code/verstehen: Philology and Theory of Algorithmic Source Texts* at the Universities of Basel and Lüneburg, funded by the Volkswagen Foundation (VW Foundation 2022).

Keywords: Source Codes, algorithms, code as a form

Autor·in

Simon Roloff
Leuphana Universität Lüneburg