Menschen ordnen Code

Zum Verhältnis von formalisierter Software und ihrer sozialen Herstellung

Roman Tischberger

Wenn ich in einem Museum bin und dort einen Jaquard-Webstuhl erblicke, freue ich mich immer doppelt. Zum einen, weil die reine Optik aus all den Hebeschnüren überwältigend ist. Über millimetergenau ausgeklügelte Führungen entfaltet sich eine kunstvolle Choreografie textiler Stränge, die in ihrer imposanten Symmetrie eine Ordnung herstellen, welche schnelles und präzises Steuern der Stellungen einzelner Kettfäden erlaubt – und damit komplexe Muster in kürzester Zeit entstehen lassen kann. Zum anderen, da Webstühle und gewebte Muster die Idee von Code greifbar machen: Als Ausgangspunkt einer Handlungsanweisung und deren konkretes Ergebnis. Darüber hinaus empfinde ich die Automatisierung von Webmustern mittels Lochkarten als eine derart geniale Technik, dass ich angesichts der unwiderstehlichen Klarheit dieses Systems ganz demütig werde, da in den eigenen Weltdeutungen selten etwas so klar scheint.

Allein die Anordnung von Kett- und Schussfäden, deren Beschaffenheit und Färbung lassen Muster im textilen Werkstück sichtbar werden. Egal ob Klassiker wie Vichy-Karo, Hahnentritt oder komplexere florale Muster: Immer liegt eine festgelegte Reihenfolge des Hebens und Senkens des Fadenwerks zugrunde. Diese Automatisierung des Webens, Hebens und Nicht-Hebens der Kettfäden setzt einen geordneten Prozess und ein eindeutiges Kommunikationssystem voraus, das algorithmisch abbildbar ist. Dieses findet sich in den Lochkarten und der darin abgebildeten Lochung, dem Code. Ein binäres System mit zwei Optionen: Loch oder Nicht-Loch, das sich auf das Heben oder Nicht-Heben eines eindeutig zugewiesenen Kettfadens auswirkt, je nach der Positionierung auf der Lochkarte. Eine Lochkarte entspricht einem definierten Umfang an gehobenen Kettfäden, hintereinander gereiht bilden die Lochkarten ein Programm für die entsprechende Musterung des Gewebes. Der verwendete Code dient so als eine im System des Webstuhls geteilte Logik, um die komplexe Kommunikation des Webstuhl-Netzwerks zwischen Menschen, Garnen, Musteranweisungen und Heben oder Nicht-Heben der Kettfäden abzukürzen und schließlich im Textil zu materialisieren.

Code, so sei aus diesem einführenden Beispiel gelesen, hat zunächst zwei Bedeutungsebenen: Einerseits als konkrete Lochkarte und die darin codierten Handlungsanweisungen, einfachen Algorithmen. Zusammen lassen die Karten sich zu einem Programm verbinden, das im Vorfeld festlegt, welche Fäden gehoben werden. Programmcode bezeichnet somit eine konkrete Zeichenfolge. Andererseits bezieht sich ‚Code' auch auf das dahinterliegende Regelwerk. Es ist die Abstraktion des eigentlichen Vorgangs, symbolisch reduziert und hier übersetzt in gelochten Karten. Es dient zur Standardisierung von Kommunikation und bietet als kollektiv geteilte Logik eine Grundlage, um Informationen eindeutig auszutauschen. Innerhalb dieses von menschlichen wie nicht-menschlichen Kommunizierenden interpretierbaren Regelwerks lässt sich eine offene Vielfalt an Kombinationen herstellen. Eine begrenzte Anzahl an Befehlen kann zu annähernd unbegrenzten Möglichkeiten kombiniert werden.

Definitionsversuche: Code jenseits des Technischen

Wie versuchen Ansätze in der Informatik das Wesen von Code zu greifen? Das Lehrbuch „Grundkurs Informatik" widmet den kommunikativen Grundlagen von Computern zwei Kapitel, um darin auch den Begriff des Codes zu erläutern.[1] Ausgehend von der axiomatischen Setzung, dass Kommunikation einen Übertragungsvorgang von konkreten Nachrichten zwischen mindestens zwei Parteien bezeichnet, trennen die Autoren die Begriffe „Nachricht" und „Information" qualitativ. Während eine Nachricht aus einer festgelegten Zeichenfolge im Rahmen eines definierten Zeichenraumes wie einem Alphabet besteht, sei die Information „ein sehr vielschichtiger Begriff, der mathematisch nicht einfach und vor allem auch nicht allgemein in all seinen Facetten fassbar ist."[2] Dies liege im semantischen Interpretationsspielraum begründet, den Information jenseits der normierten Nachricht beinhalte. Die Nachricht selbst, also das Vehikel der Information, läge dabei immer in einem codierten Rahmen vor. Entscheidend diesbezüglich sei eine umkehrbare Eindeutigkeit zwischen Sende- und Empfangseinheit – beide Nachrichtenräume müssten die Codierung verstehen. Wenn dies gegeben sei, würde die Art und Weise der Codierung zweitrangig. Codierung würde dann sowohl ein eindeutiges Symbolsystem bezeichnen als auch den Vorgang des En- und Decodierens mit einbeziehen: „Von einer guten Codierung wird man vor allem erwarten, dass sie die Darstellung zu sendender Daten mit möglichst wenigen Zeichen erlaubt und dass sie möglichst unempfindlich gegen Störungen ist."[3]

Diese abstrakte und formalisierte Definition ist jedoch nur eine Seite der Medaille. Der Begriff des Codes, konkreter des Programmcodes, ist jenseits der textuell-verwobenen Befehle eines fixierten Regelwerks oder im Sinne einer beste-

Abb. 1: Code, Gewebe und Porträt: Diese Darstellung in Optik einer Gravur zeigt Joseph-Marie Jaquard in einer Werkstatt. Die Besonderheit: Das Porträt ist aus Seide gewebt und wurde im Jahr 1839 von Didter Petit et cie in Lyon mit einem Jaquard-Webstuhl gefertigt.

henden Handlungsanweisung und deren formalisierter Rahmung zu fassen. In rein informatischen Definitionen wird oftmals vernachlässigt, dass Code zudem eine wandelbare, prozessuale, gänzlich soziale Dimension hat, worauf Adrian Mackenzie hinweist. Code ist für den Soziologen nicht nur das techno-minimalistische Kommunikations-Regelwerk und ein Bausteinbegriff des komplexeren Gewebes der Software, sondern „(it) designates a multidimensional and mutating object of analysis. It encompasses different practices of production, consumption, use, circulation and identity."[4] Um das Wesen von Software in seiner Gesamtheit zu betrachten, ist es sinnvoll, den Begriff und das damit verknüpfte informatische Verständnis von Rechenoperationen und Relationen zu de-abstrahieren und auf die konkreten Bestandteile und Praktiken zu blicken, aus denen sich das ephemere Bild von Software zusammensetzt. So zeigt sich: Die soziale Seite von Software ist mitnichten klar und formalisiert, sondern geprägt von mensch-technischen Verwobenheiten, Inkonsistenzen und Aushandlungen.[5]

Die folgenden Ausführungen fokussieren die von Mackenzie angesprochenen „practices of production", also die soziale Herstellung von Programmcode in der Sphäre von Software-Entwickler:innen. Wie sehen die sozialen Prozesse mittels derer sie Code ordnen aus? Alle genannten Beispiele sind im Rahmen eines Forschungsprojekts erhoben worden, das die Arbeitskulturen von Softwareschaffenden am Fall eines mittelständischen Software-Dienstleisters untersucht. Wenngleich das Forschungsdesign auf das gesamte Unternehmen und dessen Mitarbeitende zielt, beschränken sich die hier verwendeten Passagen auf Gespräche mit Software-Entwickler:innen. Ergänzt wird das Material durch ein Interview, das in der vorbereitenden, explorativen Phase mit einem Informatiker aus dem universitären Umfeld geführt wurde.

Programmieren als Mensch-Maschine-Interaktion

„Die Maschine ist knallhart. Die bestraft dich für alles", entgegnet mir Stefan, nachdem ich dem an der Universität angestellten Informatiker erzählte, dass ich, um Software selbst besser zu verstehen, eine Art Programmier-Bastelanleitung für den Einplatinencomputer Raspberry Pi ausprobiert hatte. Mehrere Tage grübelte ich über einem Fehler, den mir das kryptische Logfile des zu bauenden Programms ausgab. Schließlich fand ich ihn in seiner ganzen Banalität: ein fehlendes Semikolon, das einen Befehl unbrauchbar und dadurch das gesamte Programm nicht durchführbar gemacht hatte. Für Stefan eine allzu bekannte Situation. „Wenn du nen Fehler irgendwo hast, den siehst du oft gar nicht. Du überliest das." Der Computer hingegen bemerkt die den Formalien ungenügende Codezeile sofort, „jedes Leerzeichen, jedes Komma, was auch immer falsch ist. (…) Das kann's nicht mehr interpretieren."[6] Daran wird exemplarisch deutlich, wie aus einer kleinen Ungenauigkeit weitreichende Konsequenzen wachsen können.

Auch wenn Programmiersprachen eine logische und eindeutige Systematik zugrunde liegt, schließt das nicht ein, dass darin verfasster Programmcode ebenfalls eindeutig ist. Die Menschen können ihn nicht immer in der nötigen Konsequenz verfassen, umarbeiten oder überblicken, vor allem nicht mit zunehmender Komplexität eines Programms. Missverständnisse, Tippfehler, Unklarheiten, Mehrdeutigkeiten, Demotivation oder Prokrastination prägen Praktiken des Programmierens und stören das vermeintlich klare Bild von Software.[7] Jedoch ist der Wille und die Notwendigkeit zur Lösungsfindung hoch, wie es Stefan beschreibt: „Du kommst also eigentlich selten an einen Punkt, wo du sagst: Das ist generell nicht lösbar. Sondern du bist eigentlich immer an dem Punkt: Es ist lösbar, du bist nur zu dumm. Und das motiviert dich so. Frustriert auch, aber motiviert auch. Weil du halt weißt: Es muss eine Lösung geben. Und es gibt auch immer irgendwie eine."[8] So ist die Programmierpraxis eine ständige Interaktion zwischen Menschen und Maschine. Programmierende verfolgen ein Ziel, das sie in Software umsetzen möchten, und nutzen dazu eigene wie fremde Wissensbestände. Eigenes Wissen über den Aufbau von Software, fremdes Wissen wie beispielsweise in der Verwendung bestehender Codebausteine oder bei Recherchen in Online-Communities als Problemlösungsstrategie. In allen Fällen entsteht Programmcode und mit ihm ein mehr oder weniger tief verschachteltes Befehlsgeflecht aus Relationen. Um das Programmierziel zu erfüllen, muss der Code, oder auch nur Teile davon, fehlerfrei ausgeführt werden. Die Menschen sehen den Codezeilen allein nur schwerlich an, ob ein Programm fehlerfrei funktioniert oder nicht. Zu unübersichtlich, fragmentiert und dezentralisiert stellt sich Software dem Menschen dar. Ganz anders die Maschine: Sie liest nach entsprechend vorgegebener Reihenfolge und führt das Programm aus, Befehl für Befehl für Befehl. Je nach Programmiersprache und genutzten Rahmenbedingungen muss der Code durch übersetzende und prüfende Programme wie Interpreter oder Compiler laufen – und wird so an der maschinellen Instanz des Regelwerks der Programmiersprache gemessen. Fällt diese Prüfung fehlerhaft aus, so sind wieder die Menschen gefordert: Wie ist die Rück-

meldung auf dem Computerbildschirm zu interpretieren? Wo könnte der Fehler liegen? Und – je komplexer und je mehr Autor:innen der Code hat – welche Programmteile sind betroffen? Wie sind sie in die Analyse mit einzubeziehen? Und welche Änderungen ziehen welche Konsequenzen nach sich?
Um diese allzu menschlichen Hindernisse der Programmierarbeit abzumildern und einzuhegen, zeigen sich unterschiedliche Ansätze, technische wie soziale. Ein Füllhorn technischer Hilfsmittel hat sich im Verlauf der Programmiergeschichte eröffnet, die oftmals softwareseitige Möglichkeiten darstellen, die Prozesse des Fehlersuchens und -behebens zu unterstützen. Wenn auch nicht alle Fehler vermieden werden können, wird der Umgang mit ihnen durch die Hilfsmittel zumindest erträglicher. Dazu zählen ‚Klassiker' wie Logfiles zur retrospektiven Prozessverfolgung oder die Nutzung bereits bestehender Software wie Bibliotheken, die quasi als Fertigteil implementiert werden können. Nicht zuletzt sind die gegenwärtigen Programmierumgebungen mit unzähligen Tools ausgestattet, die Softwareentwicklung als eine durch Autopiloten gestützte Tätigkeit mit doppeltem Boden erscheinen lassen. Ein genauerer Blick soll nun auf jene Hilfsmittel geworfen werden, die als ‚sozial' bezeichnet werden könnten und die vor allem in der kollaborativen Programmierpraxis zum Tragen kommen.

Die soziale Ordnung von Programmcode

Bei allen technischen Standardisierungen des Programmierens bleiben die Arten und Weisen, wie eine Problemstellung in Programmcode gelöst wird, kontingent. Diese im Wesen von Code angelegte Offenheit ist Stärke und Herausforderung zugleich, wie Stefan andeutet:

> „Das ist ja das Schöne an Software, dass es auch wiederverwertbar ist. (…) Deswegen diese ganzen Open-Source-Geschichten und Leute, die sich da in ihrer Freizeit stundenlang hinsetzen und irgendwas erzeugen, was andere wiederverwenden können, wieder dran weiterarbeiten können. (…) Du kannst immer auf tausend Wegen zum gleichen Ergebnis kommen. Da gibt's nicht eine Lösung, sondern jede Lösung unterscheidet sich halt in bestimmten Punkten. Die können erstmal das gleiche Ergebnis liefern, manche können Fehler haben, aber manche dann vielleicht besser wartbar sein oder besser auf ein anderes Gerät portierbar sein."[9]

Zwei entscheidende Merkmale von Code kommen hier zur Sprache, die in ihrer Kombination in der Programmierpraxis weitreichende Konsequenzen haben: Plurale Autor:innenschaft und Multioptionalität – tausend Wege zum selben Ergebnis. Dies hat nicht nur Auswirkungen auf den Weg, der im konkreten Programmierprozess von Software gegangen wird, sondern vor allem auch auf die Anschlussfähigkeit und Erweiterbarkeit des Programms. Welche Sprachen und Bibliotheken verwendet werden, mittels welcher Variablen oder Funktionen das gewünschte Ergebnis erzielt wird – Software ist pfadabhängig. Dies impliziert, dass existierender Code von den Programmierenden durchschaut und verstanden werden sollte, wenn er erweitert oder angepasst wird, damit das Programm stabil und funktionsfähig gehalten werden kann. Solange die Codebasis eine kritische Größe nicht übersteigt und nur eine einzelne Person oder sehr kleine Teams daran arbeiten, bleibt das Geflecht überschaubar. Dem Großteil von gegenwärtigem Programmcode liegt in der Regel jedoch eine verteilte, heterogene Autor:innenschaft zugrunde. Sowohl in zeitlich isolierten Momentaufnahmen als auch im historischen Gewachsensein von Code programmieren verschiedene Entwickler:innen das Befehlsgewebe kollaborativ – mitunter ohne sich persönlich zu kennen oder überhaupt voneinander zu wissen.
„Du hast nen Stil. Und du kannst auch oft am Stil erkennen, wer was gemacht hat", meint Stefan. Natürlich würden auch die ersten Entscheidungen zu Beginn eines Software-Projekts viele Entscheidungen zum Stil vorgeben, aber „wenn jemand mit dem Code angefangen hat und das einigermaßen konsistent gemacht hat, dann musst du das so weiterführen."[10] Daher sei es zumindest in der Theorie unabdingbar, neben der Einigung und Einhaltung eines geteilten Stils auch eine parallele Dokumentation der Bedeutung von Funktionen und Codeteilen, beispielsweise in Form von Codekommentaren zu pflegen. Somit wird Programmcode sozial reglementiert, um ihn auch in komplexeren Softwareprojekten lesbar und angemessen schnell interpretierbar zu halten. Kollektive Programmierleitlinien, sogenannte Code Conventions, sollen sicherstellen, dass die Multioptionalität der auf mehrere Köpfe verteilten Programmierpraxis in einem überschaubaren Rahmen kultiviert wird.
Thomas, ein weiterer Gesprächspartner, arbeitet als Software-Entwickler in einem Projektteam, das Maschinensteuerungen in der Energietechnik programmiert. Er kennt die grundsätzlichen Probleme von Teamarbeit und Wissensverteilung:

> „Wenn ich nicht weiß, dass irgendwo was zum Teil intern schon implementiert ist, dann mach ich's vielleicht nochmal und hab die doppelte Arbeit. Und vor allem wenn der Teil dessen, was ich doppelt implementiert hab', irgendwann mal geändert werden muss, dann hab' ich eine hohe Chance, dass ich nicht alle Stellen finde, wo es implementiert ist."[11]

Die prinzipielle Offenheit im Coding, die ‚tausend Wege', die bereits Stefan zuvor ansprach, führen oft zu Unübersichtlichkeit, redundanter Arbeit, Fehlern und – um in der Gewebemetapher zu bleiben – zu ungewollten Knoten und Verfilzungen. Doch nicht nur Nicht-Wissen über bestehenden Code evoziert Hindernisse, sondern auch uneinheitliche Arbeit, wie Stefan ergänzt:

> „Eine andere Sache, die zum Teil auch damit zu tun hat, sind Inkonsistenzen übers Team oder über das Projekt, was das Naming von Dingen (…) und die Formatierung von Code allgemein angeht. Das hat von ‚Wie sehr rücke ich was ein?' bis über ‚Wie verwende ich Unterstriche?' zu tun. Und je nachdem, wie das Team und das Projekt technisch aufgesetzt ist, kann es sein, dass das Team da sehr viel manuell darauf achten muss. Und am schlimmsten ist eigentlich, wenn es zwar eigentlich Code Conventions gibt, aber

ein oder zwei Leute im Team diese nicht mögen und dann so U-Boot-mäßig in ihren eigenen Codeteilen schreiben, wie sie wollen, um sie für sich selber schöner zu machen."[12]

Individuelles wie geteiltes Wissen über den bestehenden Programmcode ist ein Schlüssel zu funktionierender Software.[13] In diesem Sinne versuchen im Team akzeptierte Code Conventions eine Norm zu setzen, um das Erscheinungsbild und damit die schnellere und eindeutigere Interpretierbarkeit der Codezeilen durch Menschen zu gewährleisten. Der Code selbst, wie ihn der Computer verarbeitet, funktioniert dadurch nicht notwendigerweise merklich schneller oder besser, doch ist er für diejenigen, die an ihm arbeiten, in seinen komplexen Verästelungen leichter zu lesen und zu verstehen. Programmcode, der einheitlich formatiert und stringenten Konventionen unterliegt, bleibt jedoch immer ein Idealzustand. Gerade in größeren Teams, die nicht nur selbst geschriebenen Code verwenden, sondern die auch auf sogenannten Legacy Code aufbauen. Dabei handelt es sich um bereits in der Software historisch bestehenden Code, den die Entwickelnden eines Projekts von ihren Vorgänger:innen ‚geerbt' haben. Dann ist das Wissen über die Gesamtheit des Codes oder explizit über jene vererbten Codeteile nur vereinzelt vorhanden und muss sich aus den verfügbaren Informationen erschlossen und im weiteren Prozess im Team verteilt werden. Entscheidend dabei sind für das Verständnis und die Interpretation, jenseits von Coding Conventions, Dokumentationen über die einzelnen Codeteile.

Mnemotechniken im Code

Das „Agile Manifest" wurde im Jahr 2001 von 17 Software-Entwickler:innen des Silicon Valley aufgesetzt und unterzeichnet. Darin wird in zwölf Prinzipien und einer knappen Präambel fixiert, was den Kern einer Softwareentwicklung ausmachen sollte, die sich als ‚agil' bezeichnet. Unter anderem findet sich in der Präambel auch die Aussage „working software over comprehensive documentation".[14] Im Gespräch weist der Entwickler Thomas auf ein Detail hin:

„Das Wichtige ist: Das heißt ‚über' und nicht nur ‚statt'. Und auch ‚über verständliche Dokumentation' und nicht ‚gar keine Dokumentation'. Das heißt, es ist total wichtig, dass man sich zusammenrauft und je enger man zusammen ist, desto leichter oder besser kann man diesen Konsens, dieses Alignment herstellen. Und dann ist auch wichtig, dass man auf angemessener Granularität festhält, was das Verständnis ist und was weiter zu tun ist."[15]

Nicht nur die gemeinsame Ausrichtung an Normen und Standards ist in der Programmierpraxis elementar, sondern darüber hinaus das Fixieren der gesetzten Standards. Auch Stefan sieht das ähnlich. Einerseits hänge die Codequalität am Einhalten eigener Standards, aber „Dokumentation gehört natürlich auch dazu. Wenn du es gar nicht dokumentierst, kann es zwar schön strukturiert sein, aber (...), wenn jemand gar nicht weiß, wo er einsteigt, dann kann er damit auch nix anfangen."[16] Dies gelte nicht nur in komplexen Team-Projekten mit langer Laufzeit: „Man versteht auch seine eigenen Sachen oft mal nach einem halben Jahr schon nicht mehr."[17]
Dokumentationen können unterschiedliche Formen annehmen, je nach Zweck und Zielgruppe. Ob ein umfangreiches Funktionshandbuch, das ausführliche Festhalten von Anforderungen über Software-Features, aktive, schriftliche Diskussionen in Code Review-Prozessen oder schlicht die Kommentierung von Codeteilen im Programmcode selbst – das Fixieren von Wissen über Code ist als eine Mnemotechnik zentral für die bessere Interpretierbarkeit des Codegewebes durch Menschen. Dennoch wird dem Dokumentieren oftmals weniger Priorität als dem Programmieren eingeräumt, da es nicht immer als Teil der Kerntätigkeit gilt: „Es ist halt auch zeitaufwändig und macht auch nicht so viel Spaß."[18]
Auch der Projektmanager Philipp, ein Kollege von Thomas, bestätigt, dass Dokumentation nicht immer den höchsten Stellenwert einnimmt. Einerseits begründet er dies mit dem bereits erwähnten Agilen Manifest, andererseits sieht er genau darin eine Ursache für die Marginalisierung von fixiertem Wissen:

„Sachen wie die Dokumentation oder ein bisschen den Überblick zu behalten oder die Architektur stabil zu machen, aber trotzdem nicht zu viel zu futureproofen, was ja auch irgendwie in der Entwicklung als ein No-Go verschrien ist, weil keiner die Zukunft so genau kennt. Das ist so das große Manko an der agilen Softwareentwicklung, dass man halt sehr dazu verleitet ist, nur bis zum nächsten Sprint zu denken und nicht weiter."[19]

Aus diesen Gründen müsste umso mehr dokumentiert werden – nicht nur um Funktionen im Code selbst, sondern auch getroffene Entscheidungen besser nachvollziehen zu können und so perspektivisch Wissen für die Zukunft bereit zu stellen. Doch dem stehen oftmals andere Prioritäten, zeitliche oder ökonomische Gründe im Weg. Es zeigt sich, dass Wissen über Code, egal für welche Zielgruppe, ein Faktor ist, der eher unterschätzt wird, obwohl er von großer Wichtigkeit ist.
Wie die Dokumentationen und Code Conventions, die sozialen Aushandlungen des Codes rund um den Jaquard-Webstuhl aussahen, lässt sich hier nur noch erahnen. Dass die verwendeten Lochkarten entsprechend ihrer Funktion gekennzeichnet und nach einem System geordnet wurden, kann zumindest als wahrscheinlich gelten. Denn so überwältigend klar die Idee binärer Codierung auch scheint: Je komplexer die darin verfassten Routinen werden, desto unübersichtlicher wird die Programmstruktur. Als menschengemachtes Befehlsgeflecht entziehen sich Software und Programmcode kontinuierlich stabilen Ordnungen, eindeutiger Fehlerfreiheit und bleiben fragil. In der Mensch-Technik-Interaktion des Programmierens, Code-Interpretierens und Problemlösens versuchen Menschen diese Komplexität und Fragilität durch mnemotechnische Hilfsmittel zu kultivieren. Sowohl die technischen als auch die sozialen Hilfsmittel wie Code Conventions oder Dokumentationen können helfen, die Konsequenzen der informatisch-strikten Formalisiertheit von Codes für die daran arbeitenden Menschen abzufedern. Indem nicht nur der Programmcode selbst, sondern auch der

soziale Erstellungsprozess dieses Codes von kollektiv geteilten Ordnungen gerahmt wird, wird er besser handhabbar und robuster.

Allerdings, so Stefan, sei dies nicht immer das Ziel: Der Entwicklungsprozess von Software sei „je nach Projekt immer unterschiedlich. Die einen dokumentieren vielleicht mehr, die anderen schreiben eher mal schnell was zusammen (...) – und es geht auch darum, vielleicht mal schnell etwas zu präsentieren, das läuft, aber vielleicht nicht schön ist. Aber das sieht ja dann nach außen keiner."[20] Auch hier zeigen die ‚tausend Wege' ihre Wirkung: Solange das Ergebnis der Erwartung entspricht, wird nicht unbedingt weiter nachgefragt oder im Detail dokumentiert – auch das ist eine soziale Aushandlung.

Anmerkungen

1 Ernst, Hartmut/Schmidt, Jochen/Beneken, Gerd: Grundkurs Informatik. Grundlagen und Konzepte für die erfolgreiche IT-Praxis – Eine umfassende, praxisorientierte Einführung. 5. Aufl. Wiesbaden 2015.

2 Dies., S. 38.

3 Dies., S. 69.

4 Mackenzie, Adrian: Cutting Code. Software and Sociality. New York 2006, S. 2.

5 Ders., S. 5.

6 Interview GP02 vom 07.02.2017, Abs. 133-135.

7 Tischberger, Roman: Computer sagt Nein: Fehlerkulturen in der Softwarearbeit. In: Groth, Stefan/May, Sarah/Müske, Johannes (Hg.): Vernetzt, entgrenzt, prekär? Kulturwissenschaftliche Perspektiven auf Arbeit im Wandel. Frankfurt a. M./New York 2020, S. 109-129.

8 Interview GP02 vom 07.02.2017, 133.

9 Interview GP02 vom 07.02.2017, 59-60.

10 Interview GP02 vom 07.02.2017, 142.

11 Interview GP18 vom 24.08.2018, 96.

12 Interview GP18 vom 24.08.2018, 98.

13 Tischberger, Roman: Wie wir coden wollen. In: Moser Johannes/Koch, Gertraud/Hansen, Lara (Hg.): Welt. Wissen. Gestalten. 42. Kongress der Deutschen Gesellschaft für Volkskunde in Hamburg 2019. Im Erscheinen.

14 https://agilemanifesto.org, abgerufen am 12.02.2021.

15 Interview GP18 vom 24.08.2018, 102.

16 Interview GP02 vom 17.02.2017, 156.

17 Interview GP02 vom 07.02.2017, 144.

18 Ebd.

19 Interview GP25 vom 07.09.2018, 60.

20 Interview GP02 vom 07.02.2017, 146.