5. Softwaregestaltung basiert auf Wissen und Kommunikation

Um die beiden Forschungsfragen der Formen und Folgen von industriespezifischer Softwaregestaltung zu beantworten, sind nicht nur die beiden oben genannten Kernprobleme der Softwaregestaltung entscheidend (softwaretechnische Interdisziplinarität und Gestaltungsmöglichkeiten). Genauso wichtig für die Praxis sind Kommunikation und Wissen: Software besteht vom Quellcode über die Softwarearchitektur, ihren Einstellungsmöglichkeiten bis zur Bedienoberfläche aus mehreren Schichten. Weil für die beteiligten Menschen eine dieser Ebenen der primäre Arbeitsgegenstand ist (z.B. für die Programmierenden der Quellcode, für die Anwendenden die Bedienoberflächen), müssen sie sich über ihre unterschiedlichen Perspektiven verständigen und ihr Wissen einbringen. Das Wissen über die komplexe Welt aus Algorithmen, Daten, Einstellungsmöglichkeiten, Schnittstellen, fachlichen Prozessen etc. vor und hinter der Softwareoberfläche reicht aber nicht aus. Die Beteiligten müssen sich im Zuge der Softwaregestaltung kommunikativ austauschen können. Für die Softwaregestaltung ist neben der verbalen Kommunikation nonverbale Kommunikation als wissensbasierte Textarbeit in Form von Arbeit an Quelltext, Spezifikationen, Anforderungen, Konzepten oder Dokumentationen notwendig. Wie das Kapitel zeigt, wurde Softwareentwicklung selbst im Laufe ihrer Geschichte in Forschung und betrieblicher Praxis mehr und mehr als Kommunikationsprozess verstanden. Somit ist nicht nur die Anwendung von Software sozial bedingt. Ihr gesamter Gestaltungsprozess ist es. Das Kapitel erläutert die Prämisse der vorliegenden Arbeit, dass anders als bei anderen sozialwissenschaftlichen Ansätzen zum Verhältnis von Mensch und Technik eine Arbeitsteilung zwischen Mensch und Software(entwicklung) besteht: Der Mensch verfügt über das Wissen, kann kommunizieren und damit Software gestalten. Softwaregestaltung ist etwas genuin Menschliches, weil dafür ein sinnhafter Bezug zu Objekten notwendig ist. Damit legt dieses Kapitel die Basis für die darauffolgenden Ausführungen, auch um zu verstehen, warum Wissen und Kommunikation zwei zentrale Begriffe und Elemente der Softwaregestaltung sind. Es geht um die materielle Basis des Arbeitsprozesses.

5.1. Technische Grundlagen: Software als Ergebnis menschlicher Textarbeit

5.1.1. Verarbeiten und verstehen: Arbeitsteilung zwischen Menschen und Maschinen

Was die theoretische Sicht auf Technik anbelangt, geht diese Arbeit von einer klaren Arbeitsteilung zwischen Menschen und Computern aus: 1. Nur der Mensch kann informiert sein, etwas wissen und kommunizieren. 2. Der Mensch ist das soziale Wesen und 3. entwickelt die Software. Der Computer weiß weder etwas, noch ist er informiert, noch stellt er soziale Beziehungen her oder schreibt autonom industriespezifische Software.

Zu 1.: Für die Person, die Daten eingibt, sind die Daten Informationen und sie braucht ein bestimmtes Wissen, um die Eingabe korrekt auszuführen. Informationen sind immer soziale Interpretationen von Daten, sie haben eine Bedeutung, sie haben Sinn. Wobei aus Informationen Wissen wird, wenn sie in einen bestimmten Erfahrungskontext eingebunden sind (vgl. Willke 1998: 162). Das ist etwas, dass ein Computer nicht kann, weil interpretieren nur Menschen können. Wie von Brödner (2014) analysiert, ist der Interpretationsmoment hervorzuheben, der zwischen maschinell ausgeführten Operationen und sozialen Handlungen eingebettet ist, zwischen zu interpretierenden Daten und maschinellen Verarbeitungen (Algorithmen). Wenn Maschinen Daten liefern und diese dann zur Steuerung und Kontrolle dieser Maschinen dienen, dann müssen diese Daten von den Beschäftigten interpretiert werden (wie bspw. von Zuboff 1988 ausführlich beschrieben).

Der Begriff des Wissens markiert den Übergang zum Handeln. Mit Wissen können Menschen Probleme lösen. Mit Wissen können sie in einem bestimmten Kontext handeln und entscheiden. Somit ist es irreführend, davon zu reden, dass Computer handeln, entscheiden oder etwas wissen. Das können nur Menschen, die einem Zeichen eine Bedeutung beimessen können (s.o.) und dann der Bedeutung gemäß handeln.

»Wissen als Erklärungszusammenhang für Informationen, als eine mit Erfahrung, Kontext, Interpretation und Reflexion angereicherte Form der Information, geeignet, Arbeitshandeln und Entscheidungen anzuleiten« (Jürgens 1999 nach Wilkesmann 2005: 56).

Also: Wenn ein Mensch Daten Sinn geben kann, sind es Informationen. Erst wenn dieser Mensch daraus Handlungen und Entscheidungen ableiten kann, wird es zu Wissen. Genau dieser Prozess, den eine Autorin als De- und Rekontextualisierung beschreibt (vgl. Degele 2000: 69), passiert täglich in den softwaregestützten Organisationen. So z. B. wenn jemand vor einer Eingabemaske steht, die er nicht versteht, weil ihm das Wissen fehlt, oder wenn der Rechnungsbeleg alle notwendigen Informationen enthält, die/der neue Sachbearbeitende aber nicht genau weiß, warum die Rechnung nun so und nicht anders aufgebaut ist. Für die Organisation ist es wichtig, dass die Software das richtige Rechnungsformular erzeugt. Womöglich ist die Umsetzung auch dokumentiert. Für neue Sachbearbeitende wäre es jetzt wichtig zu wissen, wo sie diese Dokumente finden oder wer ihnen sagen kann, warum etwas wie auf dem Rechnungsformular gestaltet ist. Selbst bei programmiertem Quellcode ist es wichtig zu wissen, warum etwas wie entwickelt wurde. Die Bedeutung ist sonst nicht ersichtlich. Selbst eine umfangreiche Dokumentation reicht oft nicht aus, um das gesamte Wissen zu hinterlegen (vgl. D'Adderio

2003: 326). Der Quellcode kann der Maschine eindeutige Befehle geben, liefert aber nicht automatisch seine Entstehungsgeschichte und seinen Sinn für den Kontext, für den er existiert.

Um Daten, Informationen und Wissen unter Menschen auszutauschen, ist Kommunikation notwendig. Dadurch, dass der Computer Kommunikation von ihrem Kontext entkoppeln kann und wie ein Buch, ein Brief oder ein anderes Schriftstück vom Sendenden abstrahiert, verändert sich das Verhältnis von Information, Mitteilung und Verständnis durch Arbeiten via Software (vgl. Degele 2000: 65). Der Sinn der Kommunikation ergibt sich nicht mehr direkt aus der Mitteilung, sondern der/die Empfänger:in kann unabhängig davon interpretieren und der Mitteilung einen Sinn geben (vgl. Esposito 1993: 351f.). Sie/er kann aber auch daran scheitern, weil er/sie z.B. einen Begriff nicht versteht. Das alles macht menschliche Kommunikation zum wesentlichen Bestandteil der Softwaregestaltung, die der Computer nicht vollständig ersetzen kann. Das zeigt sich, wie im Folgenden dargestellt, vor allem im Anforderungsmanagement. Letztlich sorgen die Beschäftigten in einem stetigen Kreislauf aus Daten, Information, Wissen und Kommunikation dafür, dass Organisationen Software anwenden, programmieren und gestalten.

Zu 2.: Neben der interpretierenden Funktion des Menschen sind die von ihm verwendeten Zeichen Teil einer sozialen Welt. Der vorliegenden Arbeit liegt eine klare Unterscheidung zwischen Sozialem und Technischem zugrunde. Sie folgte dabei ausgehend von C. S. Peirce und Jürgen Habermas den Autor:innen Mingers und Willcocks (2014), die von drei Welten ausgehen:

A) Der Welt der Person, welche Zeichen und Nachrichten erzeugt und interpretiert (Softwaregestaltende, -anwendende, -programmierende).
B) Der materiellen Welt, in der die Zeichen verkörpert sind und übertragen werden (Software, Hardware).
C) Der sozialen Welt, weil die individuelle Nutzung des Zeichens nicht über das Soziale hinausgehen kann (z.B. der kollektive Arbeitsprozess der Softwaregestaltung, die Arbeitsteilung zwischen Anwendung und Entwicklung).

Für Mingers/Willcocks sind die oben aufgeführten drei Welten ontologisch und epistemologisch getrennt. Wobei für sie das Individuum im Mittelpunkt steht: »communications and information systems rest on individuals who create and send, or have sent, messages and data; then receive and interpret them; then act (or not act) upon them« (Mingers/Willcocks 2014: 50). Das Subjekt vermittelt zwischen materieller bzw. technischer und sozialer Welt, indem es Zeichen deutet. Damit grenzen sie sich von Ansätzen wie jenem der *Sociomateriality* ab, für den Soziales und Technisches nicht trennbar sind. Einer dieser Ansätze ist die Actor-Network-Theorie: Diese vernachlässigt für Mingers/Willcocks sowohl die vermittelnde Funktion des Einzelnen als auch die ontologischen Unterschiede zwischen Technik (Software) und der sozialen Welt (bspw. einer Organisation). Wie sehr die oder der Einzelne als Teil einer sozialen Welt bei der Softwaregestaltung agiert, führt 5.2 weiter aus und ist zentraler Bestandteil dieser Untersuchung (vor allem beim Arbeitsprozess der Softwaregestaltung selbst).

Zu 3.: Wenn eine Verwaltung bestehende Formulare in Software übersetzt und nun papierlos arbeitet, ohne dass sich etwas am bürokratischen Ablauf oder den Formularen wesentlich verändert: Entstehen hier neue Informationen oder wird hier nur etwas in Software überführt? Informatisierung, wie den Begriff unterschiedliche Autor:innen verwenden (vgl. Baukrowitz et al. 2006, Boes et al. 2018, Ziegler 2020), unterscheidet die vorliegende Arbeit von jener Tätigkeit, bei der Menschen die reale, analoge Welt in digitale umwandeln: der Softwareentwicklung.

5.1.2. Konkret und abstrakt: mehrere Schichten, sprachliche Strukturierung

Das Besondere an der Softwaregestaltung ist, dass die Beschäftigten während des Arbeitsprozesses mit den verschiedenen **technischen Schichten und sprachlichen Strukturierungen** der Software arbeiten müssen. Daraus erklärt sich auch die große Bedeutung von Wissen und Kommunikation, weil sich die Beschäftigten über diese unterschiedlichen Schichten und Begriffe verständigen müssen. Im Unterschied zu anderen Technologien besteht Software komplett aus Zeichen. Mit den zugrundeliegenden 0en und 1en beschäftigt sich in den EVU niemand. An ihren unterschiedlichen Erscheinungsformen kommt aber keiner mehr vorbei. Aus Arbeitssicht sind vier Aspekte zentral: Die Programmierung von (1.) Algorithmen verlangt je nach (2.) Programmiersprache unterschiedliche Fertigkeiten. Dazu gehört (3.) Softwarearbeit mit dem Medium der Sprache und Begriffen wie Architektur, Modelle oder Schnittstellen zu strukturieren. (4.) Es existiert eine Oberfläche als Medium zwischen Anwendenden, Daten und Algorithmen.

Zu 1.: Da sind zum einen die in der Software eingeschriebenen Anleitungen zur Datenverarbeitung: die Algorithmen. Sie stellen klare Vorschriften dar. Jeden formalisierbaren Sachverhalt kann die symbolische Maschine Computer ausführen. Dabei gibt es keinen Interpretationsspielraum und die Algorithmen sind durch ihre Schriftlichkeit klar definiert. Sie sind eindeutig, determiniert, unterscheidbar und allgemein (vgl. Degele 2000: 62f.). In einem Programm können Tausende solcher Vorschriften enthalten sein. Es ist dann eine Frage des Fokus, ob man eine relevante Funktionsweise (bspw. den Suchalgorithmus von Google) oder die Struktur einer Software (Methoden, Klassen, Funktionen, Befehle etc.) zugrunde legt, wenn man von Algorithmus spricht.

Zu 2.: Gebaut werden diese Vorschriften mithilfe von Programmiersprachen. Der Begriff »Sprache« sollte nicht in die Irre führen. Sie werden nicht wie menschliche Sprachen verwendet. Sie wurden als Medien entwickelt, um es den Menschen einfacher zu machen, der Maschine Befehle zu geben (anderes als bei der menschlichen Sprache gibt es keine Ambivalenz, Ironie oder Ambiguität). Softwarespezifische Sprachen wie ABAP (für SAP), funktionsspezifische wie R, objektorientierte wie C++ oder *Low-Code*-Ansätze zeigen, dass es genau darum geht: ABAP soll möglichst auch für Nicht-Programmierende leicht erlernbar sein. Einfache Abfragen und Ausgaben von Datenbanktabellen sollen bspw. auch für die in den Wirtschaftsorganisationen weitverbreiteten Betriebswirtschaftlern möglich sein. R wird für statistische Aufgaben verwendet und verfügt über die entsprechenden Befehle. *Low-Code*-Software (im Sinne von wenig programmieren) anbietende Unternehmen versprechen, dass jedes Mitglied einer Organisation eine Software entwickeln kann, weil keine komplizierte Programmiersprache gelernt wer-

den muss (bspw. FrontPage von Microsoft, um Webseiten zu erstellen). Objektorientierte Sprachen wie C++ ermöglichen es Programmierenden, Bibliotheken mit Quellcodes aufzubauen, die sie wie Bausteine in unterschiedlichen Kontexten verwenden können. Das erleichtert den für den Menschen sinnhaften Aufbau von Software und die sinnhafte Aufteilung der einzelnen Bestandteile. Es gibt immer noch sogenannten »Spaghetti«-Code, bei dem Befehl an Befehl aneinandergereiht ist, bis mehrere Tausend Zeilen dastehen, die schwer wartbar sind. Der Maschine ist das egal, für den Menschen eine Qual.

Zu 3.: Hier wurden schon einige analytische Sprünge gemacht, die für Software typisch sind: von Programmiersprachen über deren Eigenarten und deren Folgen für größere Mengen an Quellcode und Methoden, diesen Quellcode zu organisieren (bspw. in Klassen, Funktionen etc.). Wie im weiteren Verlauf mehr und mehr klar wird, gibt es eine Vielzahl von Konzepten, Begriffen und Methoden, um die Arbeit mit und am Quellcode, aber auch den Quellcode selbst zu organisieren. Die Vielfalt an Entwicklungsmöglichkeiten kontrollieren Modelle, damit die Programmierung nicht im Chaos endet. Bestimmte Formen der Programmierung, die den Quellcode strukturieren, stellen bereits eine Form der Modellierung dar[1]. Sie machen Modellierung alltäglich (vgl. Mahr 2009: 230f.). Modelle sind Ressourcen zum Speichern und Transportieren und sie sind Agenten »zur Konstruktion und Gestaltung neuer Realitäten« (ebd.). Sie spielen eine wichtige Rolle bei Erkenntnis- und Meinungsbildungsprozessen. Unterkategorien von Modellen sind bspw. Architekturen, Prinzipien der Systemgestaltung, Techniken der Abstraktion oder Prinzipien der Usability (vgl. ebd. 248). Vor allem der Modellbegriff der Architektur[2] ist mittlerweile weitverbreitet. Es gibt viele Definitionen von Architektur und laut einigen Autoren ist eine richtige Definition auch nicht möglich (vgl. Vogel et al. 2009: 49). Sie schlagen trotzdem eine vor:

> »Die Software-Architektur eines Systems beschreibt dessen Software-Struktur respektive dessen -Strukturen, dessen Software-Bausteine sowie deren sichtbaren Eigenschaften und Beziehungen zueinander und zu ihrer Umwelt« (ebd.: 49).

Für sie geht es darum, dass Software-Architektur »Komplexität überschaubar und handhabbar […] [macht] in dem sie nur wesentliche Aspekte eines Systems zeigt« (ebd. 10). Es geht um die Fundamente und tragenden Säulen einer Software (vgl. ebd.). Ob eine Firma intern etwas programmiert, es externen Programmierenden überlässt oder Bausteine aus der Cloud verwendet: Das wird schnell zu einer Frage der Architektur, weswegen auch Nicht-ITler außerhalb von IT-Abteilungen und -Unternehmen über sie sprechen. Weitere mittlerweile geläufige Begriffe wie Schnittstellen[3] oder Softwarepakete zeigen, wie strukturierungsbedürftig die Sprache bei der Arbeit mit Software ist.

---

1 Zum Beispiel die objektorientierte Programmierung.

2 Viel wichtiger als ihre Rolle bei der Modellierung ist die Softwarearchitektur bei der Untersuchung der Fallstudien im Empirie-Teil, weil sie die Organisation der Softwaregestaltung prägt. Darauf geht das nächste Kapitel gesondert ein und zeigt, welche konkreten Eigenschaften der Softwarearchitektur für die Analyse der Formen und Folgen der Softwaregestaltung relevant sind (6.4.2.2).

3 Meist nur noch APIs (Application Programming Interface) genannt.

Zu 4.: Neben den Medien zum Programmieren der Maschine gibt es noch die Medien zur Ein- und Ausgabe von Daten und Algorithmen: ob Web-Oberflächen, Eingabemasken, Computerspiele oder Textverarbeitungsprogramme. Anders als mechanische Maschinen wie Verbrennungsmotoren, Leichtbauroboter o. ä. erfordern sie per se keine mechanische Reaktion. Natürlich kann ein Programm so programmiert sein, dass eine Eingabe erforderlich ist oder nur eine bestimmte Zeit zur Eingabe bleibt. Software kann aber auch einfach nur Daten darstellen. Sie kann Arbeitsabläufe als Fließband darstellen – muss sie aber nicht. So oder so bleibt Software ein Medium zur Darstellung oder Eingabe von Daten. Der Mediencharakter zeigt sich bei Webseiten wie Wikipedia oder einer digitalen Zeitung. Der Inhalt ist zwar der gleiche (die Daten), aber die Aufbereitung anders, was Folgen für das Leseverhalten oder die Verbreitungsmöglichkeiten hat.

Wie die Fallstudien zeigen werden, spiegeln sich die verschiedenen technischen Schichten und sprachlichen Strukturierungen in der Arbeitsteilung zwischen Programmierenden, IT-Projektleitenden, IT-Beratenden, Key User:innen etc. wieder. In ihrer Arbeit vermitteln sie zwischen verschiedenen technischen Schichten, Perspektiven und Begriffen, wobei jeder seine Schwerpunkte hat und sie letztendlich eine gemeinsame Sprache finden müssen. Es ist Kommunikation notwendig, um die jeweiligen Perspektiven auf die Software zu integrieren und sich zu verständigen. Damit geht es bei der softwaretechnischen Interdisziplinarität nicht nur um das jeweilige industriespezifische und softwaretechnische Domänen-Wissen.

5.1.3. Zwischen Text und Blackbox: Grenzen der Gestaltung und des Verstehens

Für die Softwaregestaltung spielt es eine besondere Rolle, dass unterschiedliche Personen und Organisationen unterschiedlichen Zugriff und Gestaltungsmöglichkeiten bezüglich der Software haben und sich die Software stetig ändert. Nicht jede:r kann den Quellcode oder eine Datenbank einsehen oder verstehen und verändern. Im Verlauf der Entwicklung einer Software verändert sich, was die Beschäftigten noch gestalten oder worüber sie noch reden können (vor allem bei Standardsoftware). Das ist insofern wichtig, weil es Teil des Arbeitsprozesses der Softwaregestaltung ist, zu vermitteln: zwischen Teilen der Software, die als Blackbox erscheinen, und den analysierbaren; zwischen gestaltbaren und nicht mehr gestaltbaren Teilen der Software; zwischen Softwareoberflächen und einem Quellcode, die oder den man kennt oder einem fremd ist; zwischen einer Software und ihrem Umfeld, die sich beide stetig ändern und damit das Wissen über beide langfristig nicht gesichert ist.

Anders als bei anderen Maschinen oder Werkzeugen gibt es die Möglichkeit, den Zugriff auf Software genau festzulegen. Dies geschieht häufig durch differenzierte Berechtigungsstrukturen, die unterschiedliche Zugriffe auf Software und damit Daten, Funktionalitäten bis hin zum Quellcode erlauben (bspw. bei SAP, Windows oder diversen Online-Plattformen). Mitarbeitende in einem Call-Center müssen mit der Software arbeiten, die ihnen ihre Firma zur Verfügung stellt. Wenn die Software genaue Vorgaben macht, wie ein Anruf abzuwickeln ist, und bestimmte Daten anzeigt, können die Mitarbeitenden das nicht ändern. Es können auch einzelne Eingabefelder für Mitarbeitende freigeschaltet oder gesperrt sein. Andererseits gibt es formalisierte Wege für den Zugriff auf die Gestaltung von Software. Viele Firmen haben (formale) Wege, um an

bestimmte Berechtigungen zu kommen. Bei Schwierigkeiten oder Fehlern mit Software gibt es einen First-, Second-, Third-Level Support, der Anwendenden weiterhilft.

Für die unterschiedlichen Stakeholder einer Software gibt es unterschiedliche Möglichkeiten der Gestaltung und des Verstehens. So entscheiden meist wenige über die Softwarearchitektur, die langfristig weitreichende Folgen hat. Manche Softwarelösungen bieten Einstellungs- oder Anpassungsmöglichkeiten an, die vom Festlegen des Farbschemas bis hin zum Ersetzen einzelner Codestellen durch eigenen Quellcode reichen können. Neben der Architektur können die Anwendenden oftmals nichts mehr daran ändern, wie die Programmierenden den Anwendungskontext modelliert haben, auch wenn das ihre Arbeit beeinflusst (vgl. Mahr 2009: 230). In Software ist ein »objektiviertes Modell der organisatorischen Wirklichkeit« (Heidenreich/Kirch/Mattes 2008: 4) fixiert. Zudem ist das meiste Wissen, was in der Software steckt, nicht mehr außerhalb vorhanden oder kann nur durch Fachexpertise oder über öffentlich zugängliche Spezifikationen mühsam angeeignet werden. Der Computer ist für die meisten eine Blackbox (vgl. Zuboff 1988: 166). Das kann bedeuten, dass die Software Dinge tut, von denen die Anwendenden nichts wissen – wie z.B. unentdeckt überwacht zu werden, wie dies durch Software von Google oder Amazon passiert (vgl. Zuboff 2018).

> »Je umfassender und komplexer Maschinen werden, wandern Praktiken und Normen in die materielle Basis der Gesellschaft, allerdings black-boxed« (Joerges et al. 1998: 372).

Trotz beschränktem Zugriff auf eine Software und obwohl sie eine Blackbox sein kann, die nicht mehr änderbar ist, ist die oder der einzelne Beschäftigte für ihre/seine Arbeit auf das Wissen über die Software angewiesen. Das Wissen über den Anwendungskontext allein reicht nicht. Denn der Anwendungskontext existiert nur noch als einer, den die Software bereits verändert hat. Eine Autorin spricht von einem reflexiven Strukturierungsprozess: Beim Einsatz von Technik in organisationalen Netzwerken (in dem konkreten Fall geht es um Call-Center) bedeutet dies, dass sich das Verhältnis von organisationalem Netzwerk und Technikverwendung als eines der zunehmenden Durchdringung und wechselseitigen Gestaltung beschreiben lässt. Soziales und Technisches sind nur noch schwer zu trennen (vgl. Longen 2015: 120). In einer Studie zu einer ERP-Einführung ist von »durchwurstelt«, dem Eigenleben des Einführungsprojektes oder einer »unruly technology« die Rede: Es kann immer etwas Unvorhergesehenes passieren (vgl. Conrad 2017: 189f.). Das liegt für die Autorin daran, dass Organisation und Technik sich nicht mehr auseinanderhalten lassen.

> »Man hat es nicht mit zwei unterschiedlichen Entitäten zu tun – Organisation auf der einen Seite, Medien und Technologien auf der anderen –, sondern beide enthalten Elemente voneinander und haben sich in Abhängigkeit voneinander und in Abstimmung aufeinander ausgebildet.« (Conrad 2017: 12)

Die Beschäftigten denken immer nur noch im Angesicht der Software über ihre eigene Arbeit und Organisation nach. Über die Zeit (das können Jahrzehnte sein) findet eine Ko-Konstruktion von Organisation und Software durch die anwendenden Beschäftigten

und Softwaregestaltenden statt. Am Ende eines IT-Projektes existiert der Arbeitskontext nicht mehr, für den ursprünglich der Auftrag erteilt wurde, eine Software zu entwickeln:

> »Coevolution changes the context […] and building the system changes the context itself, a software development projects actively obsolesces its own contract« (Ralph 2015: 38).

Das Wissen über Anwendungskontext und Software ist nicht nur verschränkt. Es ändert sich auch stetig. Die softwareeinsetzende Organisation als solche hat mit einem permanenten Anpassungsbedarf zu rechnen. Bereits in den 80er Jahren schreibt Lehman (1980), dass sich Software permanent ändert. Seine ersten zwei Gesetze der Programmevolution beziehen sich darauf: 1. kontinuierlicher Wandel und 2. zunehmende Komplexität der Software. Mit dem Einsatz einer Software wird der oder die Anwendende Teil ihres Lebenszyklus. Dabei geht es nicht nur um einen allgemeinen Zyklus der Softwareevolution: initiale Software, Entwicklung, Betreuung, Ausphasung, Abschaltung (vgl. Masak 2006: 222). Wenn SAP auf die Cloud und die neue Version seiner ERP-Software S/4 umstellt und die Wartung für die alte Version R/3 ausläuft, entsteht der Zwang, die Software auszutauschen. Dabei gilt besonders bei individuell entwickelter Software: Wenn Anwendende, Gestaltende oder Programmierende neu in einen Anwendungskontext kommen, kennen sie die Vorgeschichte der nur für eine Organisation entwickelten Software nicht[4].

Letztendlich sind Anwendende, Gestaltende und Programmierende nicht nur Teil einer modellierten Welt. Sie werden auch Teil eines Produktzyklus, auf den sie wenig Einfluss haben – und damit wenig Einfluss auf einen Teil des Wissens, den sie für ihre alltägliche Arbeit brauchen und das sich stetig ändert.

5.2. Softwareentwicklung: vom einsamen Nerd zum kollektiven Kommunikationsprozess

Im Laufe der Zeit wurde Softwareentwicklung immer weniger zu einem rein technischen Problem, das Techniker:innen lösen. Wie bereits oben erwähnt, wurde es zu einer großen Herausforderung, die für den fremden Anwendungskontext nützliche Software zu programmieren. Dafür sind Methoden wie Scrum nützlich (Näheres weiter unten unter 5.2.4), die den kontinuierlichen, geregelten sozialen Austausch mit klaren Rollen in den Mittelpunkt der Softwareentwicklung stellen. Trotzdem konnte in der Forschung kein Konsens hinsichtlich einer Best Practice gefunden werden, die als Orientierung für die Kontrolle von Softwaregestaltung in unterschiedlichen Kontexten der Energiewirtschaft nützlich sein könnte. Vielmehr scheinen unterschiedliche Methoden Softwaregestaltung

4 Man spricht auch von Legacy einer Software (vgl. dazu Fischbach 2016: 395ff.). Manche individuell entwickelten Altsysteme von Firmen sind kompliziert, nicht wartungsfreundlich programmiert und man befürchtet unvorhersehbare Fehler bei Änderungen an ihnen. Verlassen Mitarbeitende das Unternehmen, die mit dem Altsystem gut vertraut waren (z.B. weil sie es selbst entwickelt haben), kann das der Anlass sein, stattdessen eine Standardlösung einzuführen.

zu ermöglichen, solange sie die zentrale Rolle von Wissen und Kommunikation für den Arbeitsprozess berücksichtigen.

5.2.1. Vom schnellen Reparieren zum iterativen, kollektiven Kommunikationsprozess

Einerseits würde man aus dem bisher Gesagten vermuten, dass Kommunikation wichtig in der Softwareentwicklung ist. Andererseits ist nicht verwunderlich, dass es bei einer neuen Technologie, die aus der Ingenieurs- und Mathematik-Tradition kommt, erst einmal um deren Erforschung und Entwicklung ging. Dafür waren komplexe Kommunikationsprozesse und kommunikative Fertigkeiten nicht entscheidend. Für die Programmierer hieß es in den 50ern noch »Engineer software like you engineer hardware.« (Boehm 2006: 13) Oftmals fand IT-Arbeit damals noch in Forschungs- und Entwicklungsabteilungen statt, wo die Techniker:innen unter sich waren. Unter seines/ihresgleichen sind die Wissensgrenzen geringer. Als einen extremen Typus sieht Weizenbaum die zwanghaft Programmierenden an, für die Programmieren ein Selbstzweck ist. Ihnen geht es vor allem darum, mit der Maschine zu interagieren (vgl. Weizenbaum 1978: 161). Sollen sie dann Software schreiben, die in anderen Kontexten als der Werkstatt oder dem Labor funktionieren soll, ändern sich die Anforderungen. Die Erfahrung der Beherrschbarkeit der Maschine durch Erteilen eindeutiger Befehle via Programmiersprache wird unreflektiert auf soziale Zusammenhänge übertragen, in der diese Software entsteht oder in der sie wirken soll (vgl. Klischewski 1996: 78). Die Widerständigkeit des Sozialen fand erst über die Jahrzehnte hinweg in den Methoden der Softwareentwicklung mehr und mehr Berücksichtigung.

Hieß es in den 60ern »code-and-fix«, also einfach zu programmieren, schauen, ob es funktioniert, und dann zu verbessern (vgl. Boehm 2006: 14), wurden in den 70ern die getrennten Aufgabenschritte der Anforderungsanalyse und des Designs eingeführt (siehe auch Friedman/Cornford 1989). Das ursprünglich entwickelte Wasserfallmodell sah die erst mit Scrum weitverbreiteten Mechanismen der Iterationen, Prototypen und Feedbacks zwischen den Entwicklungsschritten vor. In der Praxis wurde das Wasserfallmodell aber erst einmal als rein sequenzieller Prozess ausgelegt (vgl. Boehm 2006: 15). Softwareentwicklung wurde zum Arbeitsvorgang, in dem streng abgetrennte Phasen der Spezifikation, Programmierung, Tests und Implementierung aufeinander folgen. Kritisch wird diese strikte Trennung vor allem, weil bei komplexen Anforderungen fehlerfreies Arbeiten unmöglich ist. Eine vollständige Konzeption oder Spezifikation ist nicht möglich, weil sich u.a. die Anforderungen der Anwendenden im Projektverlauf ändern. Das kann an einem veränderten Umfeld liegen (Konkurrenzdruck, Markt verändert sich) oder daran, dass technische Möglichkeiten erst bewusst werden, dass es Kommunikationsprobleme gab oder dass erst in der Anwendung neue Ideen auftauchen (vgl. Funken 2001: 30). In einem bekannten Artikel von 1980 schreibt Lehman, dass ein Programm nie korrekt sein kann, weil es die Umwelt nicht komplett beschreiben kann. Software ist immer nur ein Modell der Welt. Für ihn kann es bei Software deshalb nicht um absolute Korrektheit gehen (was eine mathematische Herangehensweise bedeuten würde), sondern um die Relevanz der Ergebnisse oder die Anwendungsfreundlichkeit (vgl. Lehman 1980: 1064). Er führt auch eine Unterscheidung verschiedener Programmtypen ein. Das

ist insofern wichtig, weil es darauf hinweist, dass es jenseits der hier behandelten industriespezifischen Softwareentwicklung selbstverständlich weiterhin Programme gibt, die einige wenige oder sogar eine programmierende Person allein nach einer klaren Spezifikation entwickeln kann (bspw. ein:e Physiker:in, die eine Software für ein physikalisches Experiment programmiert).

Laut Lehman kann Programmierung zudem keine Fließbandarbeit sein, weil u.a. die Entwicklung nicht bereits im Vorhinein in einfach verbundene Untereinheiten zerteilt werden kann, ohne dass sie sich gegenseitig bei der Umsetzung beeinflussen (vgl. Lehman 1980: 1065). Das liegt auch an der Abstimmung zwischen der fachlichen Domäne, in der die Software laufen soll, und den Programmierenden. Um die Nutzendenpartizipation und damit die Kommunikation zur Programmierung zu verbessern, wurde seit Mitte der 70er die Methode des Prototyping entwickelt. Sie entlastet die Anforderungsaufnahme, weil das anschauliche Ergebnis als Kommunikationsgrundlage fungiert und Nutzende direkt an der Spezifikation beteiligt sind (vgl. Funken 2001: 32ff.). Wie weitgehend sich das in der Praxis mit der Zeit durchgesetzt hat, wäre zu untersuchen.

In den 80ern stellten Floyd/Keil eine Methode vor, die eine iterativ-inkrementelle Vorgehensweise und eine kontinuierliche Kommunikation zwischen programmierenden und anwendenden Beschäftigten vorsieht. Vorteilhaft ist dabei auch die geteilte Verantwortung für die Weiterentwicklung – anstatt dass sie nur bei den Programmierenden liegt, die gar nicht wissen, was die Anwendenden brauchen (vgl. Funken 2001: 36). Die/der Programmierer:in soll nicht mehr einfach Herstellende:r sein, sondern

> »Berater[:in] in Informationsangelegenheiten, welche Multiperspektivität anerkennt und umsetzt, Vielfalt und Rückkopplung sucht und zu Revisionen bereit ist« (Floyd/Keil 1983: 36 zitiert nach ebd. 37).

Das Agile Manifesto von 2001 (Scrum ist eine der agilen Methoden) führte diesen Ansatz weiter und stellte vier Kernforderungen auf:

- »Individuals and interactions over processes and tools.
- Working software over comprehensive documentation.
- Customer collaboration over contract negotiation
- Responding to change over following a plan.« (Beck et al. 2001)

Letztendlich legt das Manifest einen klaren Fokus auf einen iterativen Arbeitsprozess mit direktem, regelmäßigem Feedback. Das Wasserfallmodell setzte sich auch deswegen nicht vollumfänglich durch, weil der Druck wuchs, Software möglichst schnell auf den Markt zu bringen, und es immer mehr Software gab, bei der die Benutzendeninteraktion im Vordergrund stand. Anforderungen waren schwerer im Vorhinein festzustellen. Sie wurden emergent und folgten dem IKIWISI-Syndrom – I know it when I see it: Die anwendende Person konnte erst sagen, ob die Software den Anforderungen genügt, wenn sie das Programm selbst gesehen hat und testen konnte (vgl. Boehm 2006: 18). Deshalb war schon die Verwendung von Prototypen ein Fortschritt. Die agilen Methoden bewerkstelligten das, indem in kurzen Zyklen (bspw. monatlich) ausführbare Software erstellt

wird, die dann die zukünftigen Nutzenden oder die Fachleute testen und dazu Feedback geben können.

5.2.2. Kommunikationskompetenz und -kern: Anforderungsmanagement

Ohne auf alle agilen Ansätze[5], geschweige denn alle anderen Methoden eingehen zu können, soll der obige kurze Abriss andeuten, dass die Softwareentwicklung sich selbst erst mit der Zeit methodisch mit ihrer sozialen Einbettung befasst hat. Umfangreiche Feldstudien in den 80ern haben festgestellt, dass nicht fehlende technische Fertigkeiten das Problem waren und sind. Die Softwareproduktivität und -qualität beeinflusst vor allem

> »zu geringe und zu wenig verbreitete Kenntnisse der Entwickler über das Anwendungsgebiet, sich verändernde und widersprüchliche Anforderungen an das Software-Design und Kommunikations- und Kooperationsprobleme zwischen Entwickler und Kunden« (Funken 2001: 46).

Es wurde abgerückt davon, sich allein auf technische Fertigkeiten zu konzentrieren:

> »Software-Entwicklung und -gestaltung muß also […] in wesentlichen Teilen als ein Lern-, Kommunikations- und Aushandlungsprozeß verstanden werden, der hohe Kooperations- und Kommunikationsanforderungen – mithin soziale Kompetenzen – an die Entwickler stellt.« (Funken 2001: 48)

Mehrere Autor:innen weisen in den 80ern und 90ern drauf hin, dass das auch in der Ausbildung von Informatiker:innen berücksichtigt werden sollte (vgl. Funken 2001: 46ff., Baukrowitz/Boes/Eckhardt 1994).

> »[D]rei Viertel ihrer Arbeitszeit benötigen Software-Entwickler für die Kommunikation mit verschiedenen Partnern: Auftraggebern, Benutzern, Kollegen, Management, Vertrieb usw.« (Funken 2001: 48)

Christiane Floyd sprach 1992 von »software development as an insight-building process in terms of multiperspectivity, self-organization and dialogue« (Floyd 1992: 86) und eben nicht davon, dass Anforderungen fix auszumachen sind wie technische Eigenschaften einer Maschine oder in einem kontrollierbaren, experimentellen Setting. Anders als bspw. bei einem Labor-Experiment ist der Entwicklungsprozess nicht durch innertechnische Rationalität vorgegeben, sondern ist ein Gestaltungsprozess, bei dem nicht nur ein technisches System, sondern auch »die sozialen Zusammenhänge seiner Verwendung modelliert werden müssen« (Schulz-Schaeffer 1996: 8).

Der kommunikationsintensivste Teil der Softwareentwicklung, das Requirements Engineering (auf Deutsch meist: Anforderungsmanagement), entwickelt sich seit den 70ern zu einem eigenständigen Forschungsfeld (vgl. Funken 2001: 52). Es stellt die korrekte und objektive Darstellung von Anforderungen in Frage. Es plädiert dafür, unterschiedliche Meinungen, Perspektiven und Sichten zu berücksichtigen. Unter anderem

5 Wie Extreme Programming, Kanban, Scrum etc.

sollen auch potenziell konfliktträchtige Perspektiven aufgenommen werden (vgl. Funken 2001: 56f.). Das Schreiben von Anforderungen, die dann die Programmierenden umsetzen, hat etwas von einer Sozialforschung: Wer mit wem wie interagiert und wie die Arbeitsabläufe sind, muss erfragt und beobachtet werden. Die Anforderungsaufnahme kann Methoden wie Interviews, Ethnografie, Perspektivenübernahme oder diskursive Anforderungsanalyse verwenden. Die Anforderungsstrukturierung nutzt Skizzen, Use Cases, Diagramme etc. (vgl. Kaminski 2012: 112ff.). Im Prozess der Anforderungsaufnahme treten IT-Fachkräfte als Kommunikations- und Übersetzungsexpert:innen auf, wobei die Programmiersprache einen Eindeutigkeitsdruck auf die Kommunikation des Anforderungsmanagements ausübt (vgl. Kaminski 2012: 89). Es muss Übersetzungsarbeit auf dem Weg zum Quellcode geleistet werden, weil Anwendende, Auftraggebende und Programmierende unterschiedliche Sprachen sprechen (vgl. Kaminski 2012: 91). So entscheidend ist die Sprache dabei, dass selbst sprachliches Framing relevant ist, um zu verstehen, wie Entwicklungsprozesse ablaufen und Expert:innen Autorität gewinnen (vgl. Alvarez 2002: 103).

Die Kommunikation muss es den Systemfachleuten ermöglichen, sich mit dem fachlichen Kontext vertraut zu machen, und den fachlichen Kontextexpert:innen, sich mit der Systemsprache vertraut zu machen. Nur so kann der Formalisierungs- und Systembildungsprozess funktionieren (vgl. Kaminski 2012: 121). Anforderungen aufzunehmen ist für einige Forschende vor allem ein Meinungsbildungs- und Verbalisierungsprozess:

> »Based on this vision, much of what occurs during the requirements process should be about opinion and will formation that is, the development of an understanding of, and the creation of meaning – about the organization and its goals and processes for achieving these goals, supported by new systems« (Ross/Chiasson 2011: 134).
> »[R]equirements elicitation takes on the form of a ›confessional‹ act where the individual verbalizes thoughts, intentions and consciousness« (Alvarez 2002: 85).
> »The RE process is a socio-technical activity. It requires intensive communication among stakeholders who have different backgrounds, skills, culture, knowledge, and behavior« (Alsanoosy et al. 2020: 356).

Erfolgreicher Wissenstransfer, gegenseitiges Verständnis (gemeinsame Konventionen und Sprache) und Kommunikation sind wesentliche Faktoren für eine erfolgreiche Softwareentwicklung (vgl. Corvera Charaf et al. 2013: 117).

Das gilt ebenso bei der Implementierung einer Standardsoftware. Es geht darum, inwiefern diese anzupassen oder wie sie einzustellen ist. Auch hier müssen die Anforderungen der Kundschaft erst aufgedeckt werden, weil sie für Beratende und Kundschaft nicht so klar auf der Hand liegen (vgl. Mormann 2016: 169). Dabei haben es die Beratenden in der Hand, welche Möglichkeiten der Software sie preisgeben oder bspw. aus Kostengründen die Gestaltungsmöglichkeiten einschränken (vgl. Mormann 2016: 186).

Wie bereits oben aufgezeigt, sind Begriffe wie Funktionen, Architekturen, Modelle oder Schnittstellen Hilfsmittel, um über Software zu reden. Dabei können im Prozess des Anforderungsmanagements nicht nur einzelne Funktionalitäten eine Rolle spielen, sondern auch wie die Software aufgebaut ist. Modelle dienen dazu, um über Software zu diskutieren und sie zu dokumentieren. Sie spielen in unterschiedlichen Entwicklungs-

methoden jedoch eine unterschiedliche Rolle. Manche Methoden[6] betrachten Modelle von vornherein als vorläufig und als fortlaufend anzupassen (vgl. Mahr 2009: 245). Agile Softwareentwicklung verwirft den »Gebrauch von Modellen zugunsten unmittelbarer Programmierung« (Mahr 2009: 246). Folglich wird das Programm selbst zur Referenz, um über die gedachten Modelle zu reden und sie anzupassen. Abhängig von der Methode unterscheidet sich dann die Kommunikation im Anforderungsmanagement und damit der Softwaregestaltung.

5.2.3. Kommunikation und Wissen organisieren: Local Practice statt Best Practice

Um die Softwareentwicklung so zu organisieren, damit sie »the right thing« (Friedman/Cornford 1989: 204) tut, hat die Prüfung der Forschungsliteratur keine Best Practice zutage gefördert. Vielmehr existieren lokale Praktiken und widersprüchliche Vorgehensweisen. Daraus ergeben sich Ansätze, aber noch keine Konzepte für die Beschreibung dessen, was bei industriespezifischer Softwareentwicklung in der Phase der Softwaregestaltung in unterschiedlichen Kontexten zwischen Anwendung und Programmierung passiert.

Unabhängig von einzelnen Methoden wie Scrum oder dem Anforderungsmanagement betrachten die Autor:innen der »general theory of software engineering« (Wohlin et al. 2015) bei der Softwareentwicklung Wissen und Kommunikation als zentral. Den Kern der Theorie bildet das intellektuelle Kapital, welches aus dem Wissen der Organisation (organisationales Kapital wie Dokumentationen, Anleitungen oder der Quellcode selbst), der Fähigkeit von Individuen (Humankapital) und den Beziehungen zu den Kund:innen und Anwendenden besteht (soziales Kapital). Wobei soziales Kapital hilft, die zwei Kapitalsorten (Human, organisational) miteinander zu verbinden (u.a. um implizites Wissen – »tacit knowledge« – auszutauschen und voneinander zu lernen). Zentral ist für die Autorenschaft letztendlich die Kommunikation:

> »Software system development is more of a communication problem than a technical problem« (Wohlin/Smite/Moe 2015: 231)

Wie eine Entwicklungsaufgabe umgesetzt wird, hängt vom intellektuellen Kapital und dem angestrebten Performance-Ziel ab. Das heißt, die Theorie sieht durchaus vor, dass bspw. das intellektuelle Kapital nicht ausreicht, um die Aufgabe umzusetzen. Aufgabe des Managements ist es dann, die Ziele zurückzuschrauben. Eine Best Practice oder spezifische Methode schlagen die Autor:innen nicht vor. Sie haben ein situatives Verständnis von Softwareentwicklung, wobei Wissen, die Kompetenzen der Mitarbeitenden, Kommunikation und gute Beziehungen eine zentrale Rolle spielen. Diese Abkehr von einzelnen Methoden und die Hinwendung zu abstrakteren Zusammenhängen vollzieht bereits ältere Literatur. In Bezug auf Managementstrategien zur Softwareentwicklung seien keine eindeutigen Best Practices auffindbar:

6 In diesem Fall RUP (Rational Unified Process).

> »Policies [of management strategies] pursued depended on the particular task at hand, and on the particular skills, experience levels and even personalities of the staff involved« (Friedman/Cornford 1989: 358).

Die Autoren lehnen bspw. Aussagen von anderen Forschenden ab, die Dequalifizierung (»deskilling«) und direkte Kontrolle oder eine Mischung von »Slack« und direkter Kontrolle allgemein als beste Strategie ansehen (vgl. ebd. 356). Andere Forscher stellen ebenso die

> »lokale Praxis einer inkrementellen Anpassung von Vorgaben, Zielen und Vorgehensschritten an sich wandelnde oder erst spät erkennbare Erfordernisse« (Schulz-Schaeffer 1996: 1)

fest. Ein anderer Autor spricht bei Softwareentwicklung von »Zonen iterativer und kommunikativer Verständigungsprozesse« (Peter 1993: 423), weil nicht vorweg geplant festgelegt werden kann, wann wer über was kommuniziert. Trotzdem seien Entwicklungsmethoden nicht überflüssig. Sie haben einen Wert für den Prozess, weil sie Sicherheit erzeugen. Sie helfen, den Planungsprozess erst einmal in Gang zu setzen und den Beteiligten eine gewisse Handlungssicherheit zu geben (vgl. Schulz-Schaeffer 1996: 15).

Es gibt Fälle, die bestimmten Managementstrategien klare Grenzen aufzeigen. Eine Studie zeigt, wie bei ausgelagerter Softwareentwicklung autoritäre Kontrolle verhindert, dass sich ein gemeinsames Verständnis entwickelt und eine ausreichende Kommunikation stattfindet. Beides wird erst wieder durch vertrauensvolle Beziehungen möglich (vgl. Gregory/Beck/Keil 2013: 1226). Auch frühe Autoren argumentieren, dass nicht einfach mehr Arbeitskräfte produktiver sind, sondern dass Kommunikation entscheidend ist und dass erst über diese nachgedacht werden sollte (vgl. Conway 1968: 31). Bei einer Untersuchung von Softwareprojekten stellt die Autorin fest, dass Demokratisierung die Produktivität einer individualisierten, wissensspezialisierten Belegschaft steigern kann (vgl. Müller 2010: 52f.). Kooperatives Arbeiten sei vielversprechender – ob durch kooperative Planung, Eigeninitiative etc. (ebd. 281f.). Eine Studie zu globalen Softwareprojekten kommt zu dem Schluss, dass nicht mehr das Management das Wissen zentralisiert oder verwaltet, sondern die eingesetzten »[K]oordinationsmechanismen zu Wissensmanagementinstrumenten« (Kotlarsky/Van Fenema/Willcocks 2008: 99) werden. Wenn die Fachleute der Anwendungsbereiche und der Programmierung einer Software zusammenarbeiten sollen, sind statt einer vertikalen Integration oder Märkten Netzwerke die optimale Organisationsform. Das zeigt eine Studie zur Entwicklung eines digitalen Kontrollsystems für Flugzeugtriebwerke (vgl. Brusoni/Prencipe/Pavitt 2001: 610). Setzt sich also das agile Arbeiten mit seinen Kernforderungen durch?

> »The most efficient and effective method of conveying information to and within a development team is face-to-face conversation. […] The best architectures, requirements, and designs emerge from self-organizing teams.« (Beck et al. 2001)

Empirisch existiert ein gemischtes Bild, was die Organisation der Softwareentwicklung anbelangt. Selbstorganisation und direkte und offene Kommunikation sind in effizienz-

getriebenen Organisationen nicht immer vorzufinden: Autoren sehen Fälle, in denen Softwareentwicklung so strukturiert und organisiert werden kann, dass sie »dem Ideal eines strikt vorgeplant-arbeitsteiligen Arbeits- und Produktionsprozesses recht nahekommt« (Schulz-Schaeffer/Bottel 2018: 102). Zugleich räumen sie ein, dass das nicht immer so ist und Softwareentwicklung auch in teamförmigen Abstimmungsprozessen à la Scrum stattfinden kann (vgl. ebd.). Andere Untersuchungen zeigen, dass Scrum nicht automatisch zur Selbstorganisation führt (vgl. Pfeiffer/Sauer/Ritter 2014, vgl. Boes et al. 2018). Unterschiedliche Organisationsformen sind auch bei internationaler Softwareentwicklung zu finden: Zwei Fallstudien stellen einmal eine Industrialisierung und fabrikmäßige Arbeit fest und einmal eine wenig formalisierte, auf eigenverantwortliche Kommunikation setzende Arbeitsweise. Beides sind somit Beispiele für einmal direkte und einmal permissive Kontrolle in der Softwareentwicklung (vgl. Feuerstein 2012). Bei einer anderen Fallstudie hatte allein die Verlagerung der Softwareentwicklung innerhalb Deutschlands »ausgeprägte Tendenzen der Spezialisierung, Abschottung, Verlust an Aufgabenvielfalt und Zuname der Dokumentations- und Kontrollarbeiten« (Flecker/Holtgrewe 2008: 321) zur Folge. Die geringe Formalisierung der geografisch verteilten Arbeit wurde zum Problem. Sie wurde dann stärker standardisiert (vgl. ebd.). Ganz zu schweigen davon, dass es unterschiedliche Vertragsverhältnissen für Programmierende inkl. Selbstständige gab (vgl. ebd. 316f.). Daneben wirken sich die Projektphase oder die Teamgröße auf die Arbeit in der Softwareentwicklung aus: Spätere Phasen in einem Projekt sind strukturierter (Heidenreich/Kirch/Mattes 2008: 13) und bei größeren Teams werden formale Organisation und Dokumentation wichtig (vgl. Ralph 2015: 35). Allein der Typ der Software kann weitreichende Folgen für ihre Entstehung haben, wie eine Gegenüberstellung von Carmel und Sawyer (1998) zeigt: Ob eine Softwarefirma entwickelt oder intern in eine Firma selbst: Laut den Autoren besteht intern eine Matrixorganisation und es läuft bürokratischer ab. Die Softwarefirma arbeitet dahingehend u.a. selbstorganisierter und die Prozesse haben eine geringere Reife. Zudem ist die Realität vieler Softwareentwickelnde, dass sie bestehende Standardsoftware anpassen und deshalb viel Zeit damit verbringen, diese zu beurteilen, zu verändern und andere Lösungen zu integrieren (vgl. Boehm 2006: 21).

Womit wir wieder am Anfang dieses Absatzes angekommen sind: Der Ansatz von Wohlin et al. (2015) ist insofern zielführend, weil er nicht auf feste Methoden oder Managementstrategien wie Standardisierung oder Selbstorganisation setzt, wenn es um die Analyse von Softwareentwicklung geht. Zudem geht es bei der Arbeit hier um jene Phase der Softwareentwicklung, bei der die verschiedenen Kontexte eine noch größere Rolle spielen dürften und ein Wissensaustausch schwieriger zu standardisieren und kontrollieren ist. Um in der Scrum-Begrifflichkeit zu sprechen, geht es hier um die Rolle Product Owner. Sie ist für das Schreiben von Anforderungen zuständig und wie sie an ihre Infos über die Anwendung (in unterschiedlichen Industrien, Firmen oder Abteilungen) kommt. Sie findet sich in unterschiedlichsten Kontexten wieder.

Die Spezifikation einer Software kann auf unterschiedlichen Wegen gelingen. Austausch von Wissen und Kommunikation sind flexibel. Ein Meeting, eine gut formulierte E-Mail, ein klärendes, persönliches Gespräch oder klare Abläufe via Ticketsystem sind allesamt Wege, Anforderungen zu spezifizieren (mehr dazu siehe unter 6.4.4). Es bleibt die Spannung zwischen offenem und direktem Wissensaustausch und Effizienz und Wett-

bewerb, wie sie allgemein für Wissensarbeit typisch ist. Bei ihr basiert Effizienz darauf, »Wissen und Expertise als Rohstoff umzuformen« (Willke 1998: 166). Das Konzept des soziotechnischen Netzwerkes im nächsten Kapitel zeigt, wie Organisationen den Arbeitsprozess der Softwaregestaltung trotzdem kontrollieren können.

Exkurs: Was sind die Bestandteile von Scrum?

Scrum ist eine Methode zur Softwareentwicklung, bei der iterativ Konzepte an Programmierende übergeben und von diesen abgearbeitet werden. Der Scrum-Prozess besteht aus Rollen, Artefakten und Meetings (vgl. Gloger 2009: 11ff.):

Scrum-Prozess	Beschreibung
Rollen	
Product Owner:in	für die Softwarelösung (das Produkt) verantwortlich; pflegt Anforderungen (Items) in eine Liste (Product Backlog) und priorisiert sie für die Programmierenden
Scrum Team	Personen, die notwendig sind, um Anforderungen in Software zu verwandeln; managt sich selbst (inkl. Arbeitsmenge); den Standards und Prozessen von Scrum verpflichtet; für die Qualität verantwortlich
Scrum Master:in	beseitigt Schwierigkeiten, Blockaden und Probleme, die das Team aufhalten; nicht weisungsbefugt, sorgt für Einhaltung Scrum-Prozess; schult Teilnehmende in ihren Rollen
Management	für Ressourcen und Richtlinien zuständig, setzt Rahmen des Scrum-Prozesses, löst von Scrum Master:in identifizierte Probleme
Artefakte	
Product Backlog	Liste mit Anforderungen (Items), je Anforderung schätzt das Team den Aufwand
Sprint	ein Zyklus (z.B. 2 Wochen), in dem Team Items abarbeitet
Sprint Backlog	Liste mit abzuarbeitenden Aufgaben für einen Sprint, wird täglich überarbeitet und aktualisiert
Meetings	
Daily Scrum	Meeting (ca. 15 Minuten) im Team, bei dem Personen sagen: Was habe ich seit dem letzten Daily Scrum erreicht? Was will ich bis zum nächsten Daily Scrum erreichen? Welche Impediments (Hindernisse) stehen mir dabei im Weg?
Sprint Plannings	Treffen für anstehenden Sprint, über Anforderungen und Ziele des im Sprint entwickelten Softwareteils; wie wird Software aufgebaut und welche Architektur soll sie haben?
Sprint Review	Treffen, bei dem das Team Funktionalität am Ende des Sprints präsentiert; Fortschritt wird anhand von »usable Software« demonstriert
Retrospektive	Treffen, in dem das Team die eigenen Arbeitsprozesse optimiert

5.3. Zwischenfazit: Softwaregestaltung als soziologisches Problem

Für die Untersuchung der Formen und Folgen von industriespezifischer Softwaregestaltung hat Kapitel 4 dargestellt, warum Softwaregestaltung zum Kern der Digitalisierung von Wirtschaft und Gesellschaft gehört. Dies gilt auch für von Software durchdrungene Organisationen, die nicht mit Software ihr Geld verdienen. Softwaregestaltung kann in unterschiedlichen Kontexten stattfinden, die auch im weiteren Verlauf der Arbeit noch eine Rolle spielen werden: durch Softwarefirmen, in digitalen Start-ups, durch IT-DL, in den EVU selbst etc. Zudem hat Kapitel 4 für die weitere Analyse grundlegende Begriffe anhand eigener Überlegungen eingeführt, die für das hier vertretene soziotechnische Verständnis moderner Organisationen stehen: softwaretechnische Interdisziplinarität und softwaretechnische Gestaltungsmöglichkeiten (bestehend aus softwaretechnischer Ausrichtung und Zuschnitt). Eine Variante der Letzteren ist der Primat der Softwareentwicklung, bei dem eine Organisation von Anfang an auf die Softwaregestaltung ausgerichtet ist (softwaretechnische Ausrichtung) und für sich eine individuelle Software gestaltet (softwaretechnischer Zuschnitt). Eine andere Variante ist, dass sich eine Organisation auf die Anwendung einer Standardsoftware konzentriert.

Um begrifflich zu klären, was bei der Softwareentwicklung der Mensch macht und was die Maschine, hat Kapitel 5 zwischen den Begriffen Daten, Informationen, Wissen und Kommunikationen unterschieden. In Abgrenzung zu anderen Theorien über das Verhältnis von Menschen und Technik folgt die Untersuchung dem kritisch-realistischen Ansatz von Mingers/Willcocks (2014). Demnach unterscheiden sich die drei Welten von Person, Sozialem und Technik ontologisch und epistemologisch voneinander und zwischen Mensch und Technik besteht eine Arbeitsteilung: Der Mensch versteht, interpretiert und vermittelt zwischen Software und Umwelt und ist dabei in eine soziale Welt eingebunden. Kommunikation und Wissen sind seine Domänen.

Um die Softwaregestaltung als Arbeitsprozess zu verstehen, hat Kapitel 5 gezeigt, dass sie und warum sie wesentlich auf Wissen und Kommunikation basiert. Das hat technologische (u.a. zeichenbasierte Technologie, mehrere technische Schichten, sprachliche Strukturierung u.a. durch Begriffe) und organisatorische Gründe (u.a. verstärkte Einbindung von Anwendenden). Es zeigt sich am historischen Wandel der Softwareentwicklung und ihrer Methoden über die Jahrzehnte hin zu einem in vielen Kontexten weitverzweigten, vernetzten, kollektiven Kommunikationsprozess.

Die Mitarbeitenden an der Softwaregestaltung machen das, was der Computer nicht kann: Sie tragen ihr Wissen bei und kommunizieren. Das müssen sie tun, wenn sie die notwendige softwaretechnische Interdisziplinarität herstellen wollen. IT-Fachleute wie Programmierende und fachliche Expert:innen wie Anwendende müssen sich austauschen. Dabei sind sie damit konfrontiert, dass Software unterschiedliche Schichten hat: Beispielsweise kennen die Programmierenden in erster Linie den Quellcode, während die Anwendenden die Bedienungsoberfläche der Software aus ihrer täglichen Arbeit kennen. Nicht jede:r hat die gleiche Perspektive auf die Software-Oberflächen bzw. spielt die Software die gleiche Rolle im Arbeitsalltag. Die Beteiligten der Softwaregestaltung haben unterschiedlichen Einblick in die Algorithmen, verstehen nicht alle Programmiersprachen oder das Gleiche unter softwaretechnischen Begriffen wie Softwarearchitektur, Schnittstelle oder Modell. In der Softwaregestaltung kann es dazu

kommen, dass es für jede Perspektive eigene Spezialist:innen gibt: für die Programmierung, Softwarearchitektur, Datenbanken, den Anwendungsbereich Prozessteil A der zu gestaltenden Software und jenen von Prozessteil B usw.

Im Zuge der Softwaregestaltung kann Software einerseits keine Blackbox bleiben und andererseits kann sich nicht jeder intensiv mit ihrem Innenleben beschäftigen. Beides wirkt sich auf Kommunikation und Wissen in der Softwaregestaltung aus. So können Berechtigungsstrukturen, die den Zugang zur gesamten Software oder einzelne ihrer Funktionalitäten regeln, es erschweren, an das notwendige Wissen zu kommen, was in der Software steckt. Dann muss möglicherweise der/die Expert:in der Softwarefirma oder ein:e IT-Berater:in hinzugezogen werden. Zudem haben Beteiligte nicht immer die Möglichkeit, sich sämtliches Wissen über die Software und ihre Anwendung anzueignen, wenn sich die Software stetig verändert, der gewohnte Arbeitskontext sich durch eine neue Software oder ein Softwareupdate verändert hat oder die Software im Laufe der Zeit immer komplexer geworden ist. Software ändert sich oftmals stetig. Das Wissen über sie kann schnell veralten. Das liegt auch daran, dass eine Standardsoftware einen Lebenszyklus hat. Die anbietende Softwarefirma löst alte Versionen ab, ohne auf die Zustimmung sämtlicher Anwendenden zu warten. All das hat Folgen für den kommunikativen Austausch und Wissenstransfer: die verschiedenen Perspektiven und Wissensstände der Stakeholder:innen einer Software, welche Veränderungen durch wen überhaupt an der Software oder an der Organisation möglich sind, stetige Änderungen an der Software oder gar die Ablösung einer Software durch eine neue Version.

Historisch betrachtet wurde es über die Jahrzehnte immer wichtiger zu berücksichtigen, dass Softwareentwicklung kein rein technisches Problem ist (vgl. Friedman/Cornford 1989, Funken 2001, Boehm 2006). Statt sie wie Fließbandarbeit zu strukturieren, haben die Autoren des agilen Manifests 2001 einen Gegenentwurf zu dieser Arbeitsorganisation veröffentlicht (vgl. Beck et al. 2001). Das Forschungsfeld des Anforderungsmanagements der Informatik zeigt, dass Kooperation, Kommunikation und soziale Kompetenzen wichtig für die Softwareentwicklung sind. Sprache, Übersetzungsfähigkeit zwischen IT- und energiewirtschaftlichen Fachleuten, intensive Kommunikation und gar die Anwendung sozialwissenschaftlicher Methoden zur Anforderungsaufnahme stehen im Vordergrund (vgl. Alvarez 2002, Ross/Chiasson 2011, Kaminski 2012, Corvera Charaf/Rosenkranz/Holten 2013, Alsanoosy/Spichkova/Harland 2020). Diese Ergebnisse stellen eine erste Grundlage für den Arbeitsprozess der Softwaregestaltung dar. Doch berücksichtigen sie keine arbeits- und organisationssoziologische Literatur, die Softwareentwicklung in unterschiedlichen Kontexten untersucht. Eine kurze Aufarbeitung dieser Literatur konnte zeigen, dass es nicht die beste Methode oder Organisationsform für alle Fälle gibt. Stattdessen existieren lokale Praktiken und die Empirie zeigt unterschiedliche Organisationsformen der Softwareentwicklung. Ein allgemeines Konzept, um die Softwaregestaltung arbeitssoziologisch zu analysieren, wäre aber hilfreich. Die allgemeine Theorie der Softwareentwicklung ist der Versuch, ein solches allgemeines Konzept aufzustellen (vgl. Wohlin/Šmite/Moe 2015). Sie berücksichtigt die Kontextabhängigkeit von Softwareentwicklung und legt sich nicht auf bestimmte Organisationsstrukturen, Abläufe oder Managementmethoden fest. Indem sie aber organisationssoziologische und arbeitssoziologische Erkenntnisse unterschlägt,

fehlt die Berücksichtigung von unterschiedlichen Arbeits- und Organisationskontexten und wie sich diese auf die Softwareentwicklung auswirken.

Mit dem Konzept der softwaretechnischen Netzwerkarbeit, was das nächste Kapitel entwickelt, lässt sich der aus der Empirie der Fallstudien entwickelte Analyserahmen besser konzeptionell in die Forschungslandschaft einbetten. Das Konzept ist auf die Phase der Softwaregestaltung zugeschnitten, berücksichtigt die für die Softwaregestaltung wesentlichen Kontextfaktoren und ist zugleich so allgemein, dass es sowohl agile wie auch weniger agile Organisationsformen abdeckt.