Das Computerprogramm als Erfahrungsgegenstand

Jens Geisse

TECHNISCHE UNIVERSITÄT DARMSTADT

Vom Fachbereich Gesellschafts- und Geschichtswissenschaften
der Technischen Universität Darmstadt

Zur Erlangung des akademischen Grades
Doctor philosophiae (Dr. phil.)

Dissertation von
Dipl. Soz. Jens Gunnar Geisse

Erstgutachter: Prof. Dr. Alfred Nordmann
Zweitgutachterin: Prof. Dr. Sophie Loidolt

Darmstadt, 2019

Geisse, Jens : Das Computerprogramm als Erfahrungsgegenstand
Darmstadt, Technische Universität Darmstadt,
Jahr der Veröffentlichung der Dissertation auf TUprints: 2019
URN: urn:nbn:de:tuda-tuprints-88256
Tag der mündlichen Prüfung: 26.06.2019

Die Dissertation ist von mir mit einem Verzeichnis aller benutzten Quellen versehen.

Ich erkläre, dass ich die Arbeit – abgesehen von den in ihr ausdrücklich genannten Hilfen – selbstständig verfasst habe. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen. Bei der vorliegenden Dissertation stimmen schriftliche und elektronische Version überein.

Darmstadt, 10.02.2019

Inhaltsverzeichnis

1. Einleitung

Was ist eigentlich ein Computerprogramm? Die Frage erscheint vielleicht zunächst überraschend: Programme sind in unser Handeln eingebunden, ohne dass wir eine Begriffsbestimmung vornehmen müssten. Sie ermöglichen bestimmte Handlungen, treten als Nebenbedingungen auf und stellen Bezugspunkte und Objekte alltäglichen Handelns dar. Zumindest eine intuitive Vorstellung davon, was Programme sind, ist bereits durch diese Berührungspunkte notwendig.

Diese intuitive Vorstellung reicht jedoch für eine technikphilosophische oder techniksoziologische Analyse der Wechselwirkungen zwischen Programmen, (weiteren) technischen Gegenständen, Menschen und gesellschaftlichen Prozessen nicht aus. Im Folgenden soll dies beispielhaft anhand von Handlungen, die sich auf Programme beziehen, aufgezeigt werden. Die Problematik, Programme als Gegenstände klar zu erfassen, sie zu begreifen, wird beim Vergleich je zweier solcher Handlungen deutlich.

Als alltägliche Handlung können Menschen Programme bedienen, kaufen, kopieren, installieren, löschen, den Umgang mit ihnen erlernen, sie kritisieren usw. Aus Entwicklungssicht gibt es Handlungen wie Entwurf, Programmierung, Testen, Verifikation, Debugging, Refactoring usw. Gemeinsamkeit dieser Tätigkeiten ist, dass ein Programm als Gegenstand der Handlung fungiert. Konkretisiert man nun diese Beispiele, ergibt sich jedoch das Problem, dass bei den Handlungen paarweise zum Teil unklar ist, ob sie sich überhaupt auf dasselbe Objekt beziehen können:

Beispiel 1: Das Kopieren und Erlernen

Ich kopiere ein Programm auf meinen PC und lerne, damit umzugehen. Beziehen sich diese beiden Handlungen auf denselben Gegenstand? Das Kopieren ist zunächst ein computergesteuerter physischer Vorgang, bei dem z.B. die Magnetisierung einer Festplatte so manipuliert wird, dass die vorhandene Hard- und Software bei einem späteren Lesevorgang das gewünschte Programm reproduzieren kann. Das Programm ist auf der Festplatte gespeichert. Mein anschließender Lernprozess bezieht sich nun einerseits mittelbar auf ebendiese Magnetzustände, da diese es erst ermöglichen, das Programm an meinem PC auszuführen. In dieser Hinsicht hätten Lernprozess und Kopie denselben Gegenstand. Andererseits geht der Lerngegenstand „Programm“ über die greifbare Speicherung auf der Festplatte deutlich hinaus – ich könnte anschließend das Programm auch an einem anderen Rechner bedienen, auch an einem, der keinen magnetischen Permanentspeicher hat. Eingeschränkt kann ich in vielen Fällen auch andere Versionen des Programms bedienen – also Programme, die streng genommen eine andere Funktionalität haben.

Beispiel 2: Kauf, Download, Kritik und Löschen eines Programms

Ich kaufe ein Programm im digitalen Vertrieb und lade es herunter. Nach einem Testlauf merke ich, dass es häufig abstürzt, und lösche es wieder. Was sind hier die Gegenstände meines Handelns? Beim Kauf wird bereits im Alltag häufig sprachlich unterschieden: Nicht das Programm selbst ist Gegenstand des Geschäfts sondern bestimmte Rechte der Benutzung. Die Intention der Handlung ist jedoch ähnlich der bei anderen Gegenständen, die zur späteren Verwendung gekauft werden. Der Download ist ein Spezialfall

der bereits genannten Kopie – der infrastrukturell besonders voraussetzungsreich ist und keine zusätzlichen Datenträger benötigt. Beziehen sich Kauf und Download auf dasselbe Programm? Zunächst erwerbe ich mit dem Kauf das Recht und die Möglichkeit eines Downloads. Allerdings gilt das Recht auch, wenn der Download von einem anderen Server auf einen anderen PC stattfindet – z.B. weil ich verreist bin. Es gilt auch für neuere Versionen des Programms, die sich funktional unterscheiden. Dadurch, dass ich das Programm lösche, wird meine Kritik an den Abstürzen nicht gegenstandslos. Eine neue Version, die die Fehler behebt, ist ein anderer Gegenstand als das Programm, welches ich kritisiert habe, und auch ein anderer Gegenstand als das Programm, das ich gelöscht habe. Trotzdem können wir hier im Alltag von einem Programm reden.

Beispiel 3: Anwendung und Debugging

Bei der Anwendung eines selbst geschriebenen Programms fällt mir ein fehlerhaftes Verhalten auf. Ich suche im Quellcode nach dem Fehler. Suche ich also in dem Programm, das ich verwende, nach Fehlern? Hier wird bereits zwischen dem Programm und dem Quellcode zum Programm unterschieden. Trotzdem kann ich im Quellcode nach einem Fehler suchen, der erst zur Laufzeit auftritt. Es wäre auch nicht falsch, „Programm" sowohl auf die Laufzeit als auch auf den Quellcode zu beziehen – nicht zuletzt ist der Quellcode der Gegenstand, an dem ich mich mit der Tätigkeit des Programmierens abarbeite. In diesem Fall wird das Programmieren erst dadurch sinnvoll, dass ich mich auf eine spätere Übersetzung und Anwendung beziehe. Komplexer wird das Problem, wenn der Fehler nur unter bestimmten Bedingungen auftritt, z.B. nur auf Mehrkernprozessoren. Ist das Programm zur Laufzeit auf einem Einkernprozessor fehlerhaft, auch wenn der Fehler gar nicht auftreten kann? Neben der spannenden Frage, was ein Fehler genau ist, wird hier deutlich, dass sowohl die Unterscheidung zwischen Quellcode und Programm zur Laufzeit als auch der Zusammenhang zwischen diesen Gegenständen Bedingung für mein Handeln sind.

Edgar G. Daylight greift in seinem Buch *Turing Tales*[1] die Unklarheit darüber auf, was genau ein Computerprogramm ist. Anhand zahlreicher Beispiele zeigt er, dass nicht nur die alltägliche Kommunikation über Programme, sondern auch der akademische Diskurs innerhalb der Informatik immer wieder Kategorienfehler produziert, die aus der unsauberen Trennung zwischen verschiedenen Bedeutungen von „Programm" resultieren. Hier wird zwischen drei Bedeutungen unterschieden:

> „Based on several case studies, I show that computer scientists have a tendency to slide in their discourse between *concrete physical* objects (laptops), *technical artefacts* (computer programs & computer programming languages), and *abstract* objects (Turing machines and the lambda calculus), as if there is no distinction between them. Specifically, several programmers (including myself) have asserted, but not proved, that computer programming languages *X, Y, Z* are Turing complete and have then used the alleged Turing completeness of such a technical artefact *X* to end up with an impossibility theorem about programming practice. I argue that this kind of reasoning is fundamentally flawed."[2]

Ob und inwiefern diese fehlende Unterscheidung im wissenschaftlichen Diskurs

1 Daylight, Edgar G.: Turing Tales. Lonely Scholar, 2016.

2 Ebda. S. 113

innerhalb der Informatik tatsächlich stattfindet und welche Auswirkungen sie hat kann hier nicht näher erörtert werden. Zur Beantwortung der in dieser Arbeit behandelten Eingangsfrage ist relevant, dass Daylight zwischen drei verschiedenen Kategorien unterscheidet. Er kritisiert eine unsaubere Trennung zwischen diesen und zeigt, wie diese zu Falschaussagen führen kann. Den Begriff *computer program* verwendet er, um Objekte innerhalb einer der Kategorien zu benennen.

Drei Dinge sollen hier für das weitere Vorgehen in dieser Arbeit festgehalten werden: Erstens ist die Unterscheidung zwischen verschiedenen Klassen von Objekten für eine exakte Analyse zwingend notwendig. Arbeit an einem physischen Gegenstand (z.B. PC) und Arbeit an einem sprachlichen Gegenstand (z.B. Quellcode) beziehen ihre Intention auf grundlegend unterschiedliche Sachen. Zweitens ist diese Unterscheidung nicht trivial. Wie schon in den Beispielen aufgezeigt reicht einfache Intuition nicht aus, um die verschiedenen Bedeutungsdimensionen von „Computerprogramm“ zu erfassen und sauber voneinander zu trennen. Und wie Daylight aufzeigt, führen unzulässige Verkürzungen und Vereinfachungen in der Sprache über Programme zumindest zu Missverständlichkeit. Drittens deuten die Möglichkeit zu Missverständnissen und die sprachliche Klammer, die viele verschiedene Dinge mithilfe des Worts „Programm“ beschreibt, bereits an, dass zwischen diesen unterschiedlichen Dingen ein Zusammenhang besteht.

Kommen wir auf die Eingangsfrage zurück: Was ist eigentlich ein Computerprogramm? Aus den vorausgegangenen kurzen Beispielen geht hervor, dass damit sehr unterschiedliche Dinge gemeint sein können. Eine Möglichkeit zur Beantwortung der Frage wäre eine exakte Definition, die sich aus den unterschiedlichen Dingen eine abgrenzbare Teilmenge herausgreift und die Merkmale herausarbeitet, unter denen die Abgrenzung vorgenommen werden kann. Die anderen Dinge wären unter dieser Definition kein Programm. Die oben angedeutete Abgrenzung in Anlehnung an Daylight würde z.B. ausschließlich technische Artefakte als Programm erfassen. Die so gewonnene sprachliche Exaktheit kann einem Teil der Missverständnisse entgegenwirken.

Der Beitrag, den eine solche Definition zum wissenschaftlichen Diskurs leisten kann, ist kontextabhängig. Innerhalb der Informatik wäre es beispielsweise denkbar, dass Arbeiten zur theoretischen Informatik einen anderen Programmbegriff verwenden als Arbeiten zum *Software Engineering* (Softwaretechnik). Dies liegt darin begründet, dass in den beiden Teildisziplinen unterschiedliche Gegenstände untersucht werden und damit unterschiedliche Teilbereiche dessen, was man alltäglich als Programm bezeichnen könnte, relevant sind.

Das Forschungsinteresse dieser Arbeit begründet sich in der Technikphilosophie und Techniksoziologie. Ziel ist es, einen Begriff zu entwickeln, mit dessen Hilfe die Untersuchung wechselseitiger Bedingtheiten zwischen Mensch, Technik und Gesellschaft an analytischer Schärfe gewinnt. Die Schwierigkeit einer solchen Begriffsbestimmung liegt nun in der Vielfalt der (für diese Untersuchung) relevanten Gegenstände zusammen mit der Notwendigkeit klarer sprachlicher Konventionen, um ebendiese relevanten Gegenstände zu erfassen. Für die Wechselwirkung zwischen z.B. Smartphones, Kryptographie und Kommunikationsverhalten sind Programme sowohl als mathematische Objekte als auch als sprachliche Objekte, die programmiert werden, relevant – und nicht zuletzt spielen physische Eigenschaften eine wichtige Rolle, damit ständige Verfügbarkeit sicherer Verschlüsselung im wörtlichen Sinne tragbar wird.

Wie kann sinnvoll mit dieser Vielfalt umgegangen werden? Die Menge möglicher Gegenstände, die unter „Programm“ subsumiert werden könnten und tatsächlich für techniksoziologische und technikphilosophische Überlegungen relevant sind, ist schwer

abzugrenzen. Aufgrund der großen Heterogenität dieser Gegenstände sind definitorische Merkmale kaum abschließend festzulegen. Um diese alle zu erfassen wäre der Ausgangspunkt einer Begriffsbestimmung die sprachliche Verwendung des Wortes: Ein Programm ist das, was wir unter Programm verstehen. Der Begriff bleibt hier zunächst konturlos und trägt wenig zur Exaktheit der Diskussion bei: Es bleibt weiterhin erklärungsbedürftig, was bei der jeweiligen Verwendung des Begriffs im Kontext genau unter Programm zu verstehen ist. Aus diesem Grund kann dies nur als Ausgangspunkt für weitere begriffliche Überlegungen gesehen werden: Wie kann ein Begriff, der eine große Heterogenität erfasst, zu einer inneren Struktur kommen, die zu fruchtbaren Analysen führt?

Der Vorschlag dieser Arbeit ist ein Begriff vom Computerprogramm, der sowohl die notwendige Komplexität aufweist, um die Bedingtheiten und Wirkungen programmierter Technik zu erfassen, als auch eine hinreichende sprachliche Exaktheit ermöglicht, ohne dass der Programmbegriff zugunsten aufwändiger Beschreibung der jeweils zu bestimmenden Phänomene verworfen wird. Ausgangspunkt der Begriffsbestimmung ist die wechselseitige Bedingtheit von Mensch, Technik und Gesellschaft: Ein solcher Begriff muss sich dazu eignen, die Wirkung von Programmen auf Individuen und gesellschaftliche Prozesse zu untersuchen. Gleichzeitig muss er auch eine Analyse von Aushandlungsprozessen und individuellem Handeln bei der Gestaltung von Technik – hier von Programmen – zulassen. Als Mindestvoraussetzung muss der Begriff also die verschiedenen Handlungen des Programmierens und des Anwendens von Programmen gleichermaßen erfassen.

Dieses Ziel kann die Arbeit dadurch erreichen, dass ein gesellschaftlich verfasster Programmbegriff vorgeschlagen wird. Ausgehend von Wahrnehmungsprozessen, durch die der Gegenstand „Programm“ konstituiert wird, können mehrere voneinander unterscheidbare Dimensionen des Begriffs gefunden werden. Die gesellschaftliche Perspektive greift diese einzelnen Bedeutungsdimensionen auf und analysiert, wie sie miteinander in Verbindung stehen und interagieren. Dadurch weist der Begriff die geforderte Komplexität auf, um Zusammenhänge zwischen den verschiedenen auf Programme bezogenen Handlungen abzubilden. Weiter lässt die Anerkennung mehrerer klar differenzierter Bedeutungsdimensionen auch die analytische Schärfe zu, die notwendig ist, um jeweils genau zu benennen, welcher Art die behandelten Gegenstände sind und welche Merkmale ihnen dadurch zukommen.

Die Relevanz einer solchen Begriffsbestimmung für zukünftige Überlegungen zur Technik ist groß: Erstens führt die anhaltende Entwicklung immer schnellerer Computer und immer größerer Speichermedien dazu, dass immer mehr Dinge durch Programme berechnet werden können. Die Möglichkeit sehr guter Prognosen unter Einbeziehung extrem vieler Einflussfaktoren (Ein Teil von dem, was unter *Big Data* verstanden wird) könnte einen nicht unerheblichen Einfluss darauf haben, was Menschen der Technik zutrauen. Die Programme, welche die Daten erheben, speichern, analysieren und darauf basierend Prognosen erstellen, spielen dabei eine essenzielle Rolle. Zweitens nimmt die Zahl programmgesteuerter Gegenstände (z.B. *Smart Devices*) zu. Gesellschaftliche Normen, aber auch individuelle Wünsche und wirkmächtige Utopien kommen in der Art zum Ausdruck, wie diese Gegenstände verwendet werden – und welches Verhalten den Gegenstände einprogrammiert wurde. Gesellschaftliche Aushandlungsprozesse beziehen sich dabei nicht nur auf die physisch greifbaren Gegenstände – hier sei beispielhaft auf Wiebe Bijkers Arbeit[3] zu Fahrrädern, Bakelit und der Leuchtstofflampe verwiesen, in der solche Aushandlungsprozesse um technische Gegenstände analysiert werden – sondern eben auch auf das, was die darauf laufenden Programme tun und sind. Drittens gewinnen – aufgrund autonomer Systeme – Diskussionen

3 Bijker, Wiebe E.: Of bicycles, bakelites, and bulbs: toward a theory of sociotechnical change. MIT Press, 1995.

darüber, was Technik „tun“ darf, ein neues Gewicht. Dieses „Tun“ bezieht sich dabei im Wesentlichen darauf, welches Verhalten die steuernden Programme zeigen. Die Grundfrage der Ethik (*Was soll ich tun?*) fließt mit der steigenden Nutzung von Programmen auch in mehr Diskussionen über Programme (und die menschlichen Bezüge zu ihnen) ein. Diese drei Beispiele erschöpfen bei Weitem nicht die Vielzahl an Diskursen, in denen eine Begriffsbestimmung von Computerprogrammen hilfreich wäre. Sie zeigen jedoch deutlich auf, dass ein solcher Begriff sowohl die Herstellung als auch die Anwendung von Programmen erhellen muss.

Im Aufbau dieser Arbeit spiegelt sich das Vorhaben wider, von der Ästhetik[4] ausgehend einen gesellschaftlich verfassten Programmbegriff zu entwerfen. Diese Einleitung hat als erstes Kapitel das Problemfeld aufgemacht und die Ziele der Arbeit vorgestellt. Im zweiten Kapitel wird der Erkenntniszugang zu dem neuen Begriff diskutiert. Dazu werden zunächst die erkenntnistheoretischen Grundlagen dargelegt. Ausgehend von Immanuel Kants *Kritik der reinen Vernunft* wird herausgearbeitet, wie wir über Wahrnehmungsprozesse Computerprogramme erst zum Gegenstand machen. Weiter wird die für die Begriffsbestimmung folgenreiche Frage diskutiert, ob ein Begriff von Computerprogrammen ein *reiner Begriff* oder ein *willkürlich gedachter Begriff* sein kann oder ob es ein *empirischer Begriff* sein muss. Anschließend werden mit Edmund Husserl die Objektkonstitutionen näher beleuchtet, die zu verschiedenen Bedeutungen des Begriffs führen. Im zweiten Teil des Kapitels wird die Abgrenzung des neuen Programmbegriffs zu anderen möglichen Begriffen, wie etwa *Software*, dargestellt – und erläutert, warum gerade ein Programmbegriff für die vorgestellten Ziele der Arbeit fruchtbar ist.

Im dritten Kapitel wird auf dieser Grundlage ein mehrdimensionaler Programmbegriff herausgearbeitet. Es werden vier Bedeutungsdimensionen identifiziert, die verschiedene Erfahrungszugänge zu Computerprogrammen repräsentieren. Nach Anmerkungen zum Aufbau des neuen Begriffs werden die einzelnen Ebenen vorgestellt und detailliert beschrieben: Programme können als physische, syntaktische, semantische Gegenstände und zu ihrer Laufzeit als eingebettete Gegenstände begriffen werden. Bei der Analyse zeigt sich, dass innerhalb der Bedeutungsdimensionen weitere Präzisierungen sinnvoll sein können, um in der Begriffsverwendung – je nach Intention – eine höhere Exaktheit zu erreichen. Die Zusammenhänge und Interaktionen zwischen den Bedeutungsdimensionen werden mithilfe von Bruno Latours Aktor-Netzwerk-Theorie untersucht. Die Möglichkeit, Programme und auch einzelne Bedeutungsdimensionen des Begriffs als Aktanten zu beschreiben und je nach Analyseziel zu figurieren, trägt entscheidend zum wissenschaftlichen Nutzen eines integrativen Gesamtbegriffs bei. Die Assoziationen zwischen den ersten vier Bedeutungsdimensionen des Begriffs stellen im Ergebnis eine fünfte Dimension dar, die den hier entwickelten Begriff wesentlich prägt. Das Kapitel schließt mit einem Exkurs, in dem diskutiert wird, inwiefern Programme als Komposition begriffen werden können: als Gegenstände, in denen die Dinge zusammenwirken.

Im vierten Kapitel wird der vorgestellte Begriff angewendet, um auf Programme bezogene Handlungen zu analysieren. Dadurch kann sowohl der analytische Nutzen der Gliederung in einzelne Bedeutungsdimensionen als auch der Erkenntnisgewinn durch die Integration zu einem Gesamtbegriff beispielhaft aufgezeigt werden. Die hier analysierten Handlungen umfassen auch typische Methoden, die bei der Entwicklung von Programmen angewendet werden – vom Testen und Debugging bis hin zur formalen Verifikation. Das Kapitel schließt mit einem zweiten Exkurs über Zeit und Programme – und zeigt dabei auf,

4 Ästhetik wird hier im weiten Sinne als Erkenntniszugang über die sinnliche Wahrnehmung verstanden. Auf die Auseinandersetzung mit Kunst und Begrifflichkeiten von Schönheit wird im Zusammenhang mit ästhetischen Bewertungen, Normen und Urteilen verwiesen.

wie der neue Begriff auch hier zur Analyse beiträgt. Das Verhältnis von Zeit und Programmen erweist sich dabei als ausreichend komplex, um weitere Untersuchungen anzuregen.

Im fünften und abschließenden Kapitel wird ein Ausblick gegeben, wie weitere Arbeiten mit dem und über den Programmbegriff aussehen können. Entsprechend der Zielsetzung dieser Begriffsbestimmung sind hier sowohl Anregungen für die Technikphilosophie als auch für die Techniksoziologie zu finden. Der Begriff eignet sich neben seiner Verwendung in theoretischen Arbeiten auch als Grundlage für empirische Untersuchungen. Insgesamt kann er Diskurse über Programme und ihre Bedeutung für menschliches Handeln bereichern, insbesondere auch in interdisziplinären Zusammenhängen.

2. Erkenntniszugang

In diesem Kapitel wird herausgearbeitet, welche Grundlagen herangezogen werden, um den neuen Programmbegriff zu entwickeln. Ziel des Kapitels ist es, über die Erkenntnistheorie zu einer Methode zu kommen, mit der aus den menschlichen Bezügen zu Computerprogrammen ein dem Vorhaben angemessener Begriff abgeleitet wird. Die Begriffsbestimmung vollzieht sich dabei in zwei wesentlichen Schritten. Im ersten Schritt werden die Wahrnehmungsprozesse, die zu Programmen als Gegenständen führen, untersucht. Davon ausgehend können die voneinander unterscheidbaren Bedeutungsdimensionen des Begriffs identifiziert werden. Im zweiten Schritt werden diese über ihre gesellschaftliche Relevanz miteinander in Bezug gesetzt.

Die Analyse der Wahrnehmungsprozesse findet unter Bezug auf die erkenntnistheoretischen Überlegungen Immanuel Kants und Edmund Husserls statt. Mit Immanuel Kant wird hier herausgearbeitet, wie die empirischen Anschauungen von Programmen mit der Begriffsbestimmung zusammenhängen. Eine Besonderheit des Programmbegriffs ist dadurch gegeben, dass viele der möglichen Anschauungen vermittelt sind, insbesondere auch technisch vermittelt. Eine weitere Besonderheit besteht darin, dass Programme grundsätzlich etwas Gemachtes sind. Dies hat Folgen für das Verhältnis vom Möglichen zum Wirklichen. Mit Edmund Husserl wird der Weg zum Gegenstand „Programm“ genauer beleuchtet. Aufgrund unterschiedlicher Intentionalität, die das Bewusstsein auf diesen im Zuge des Wahrnehmungsprozesses konstituierten Gegenstand richtet, können schließlich die unterschiedliche Bedeutungsdimensionen des Programmbegriffs gewonnen werden.

Im Anschluss wird mit Bruno Latour das Programm als gesellschaftliches Phänomen analysiert. Die gesellschaftliche Dimension des neuen Programmbegriffs arbeitet in Anlehnung an die Akteur-Netzwerk-Theorie gerade das Verbindende zwischen den einzelnen Bedeutungsdimensionen heraus. Die Zusammenführung ermöglicht es, Interaktionen zwischen Mensch und Technik und die wechselseitigen Abhängigkeiten zwischen den unterschiedlichen Gegenständen abzubilden. Gerade durch die gesellschaftlich verfasste Verknüpfung der einzelnen Dimensionen kann der Begriff sowohl die Herstellung als auch die Anwendung von Programmen gemeinsam erfassen. Durch die Darstellung von Handlungen innerhalb von netzwerkartigen Zusammenhängen wird ihre Gemeinsamkeit in der Bezugnahme auf ein Programm deutlich, auch wenn die Gegenstände sich voneinander klar unterscheiden, weil unterschiedliche Bedeutungsdimensionen berührt sind.

Die Verknüpfung von Immanuel Kants *Kritik der reinen Vernunft* mit Edmund Husserls *Ideen zu einer reinen Phänomenologie und phänomenologischen Philosophie* ist generell nicht unproblematisch. Die beiden erkenntnistheoretischen Arbeiten sind auf sehr unterschiedlichen Grundlagen errichtet: Kant greift die Strömungen des Empirismus und Rationalismus auf und verbindet sie in seiner Kritik an beiden. Bei ihm bleibt ein außerhalb des Bewusstsein stehendes Ding an sich, welches die menschlichen Sinne affiziert, sich jedoch menschlicher Erkenntnis grundsätzlich entzieht. Husserl dagegen baut die Phänomenologie rein idealistisch auf, alle Erkenntnis kann nur innerhalb des Bewusstseins erlangt werden. Dinge an sich sind Gegenstände, auf die sich das Bewusstsein in Wahrnehmungsakten intentional bezieht, die wirkliche Welt ist Bewusstseinskorrelat[5]. Diese unterschiedlichen Grundauffassungen führen jedoch in der hier verfolgten Begriffsbestimmung nicht notwendigerweise zu Widersprüchen:

5 Husserl, Edmund: Ideen zu einer reinen Phänomenologie und phänomenologischen Philosophie. Niemeyer, 1913. S. 88f.

Erstens bleibt der Vorgang der Genese von Begriffen bei Kant (die Spontaneität des Denkens) deskriptiv. Durch seine Entwicklung der Kategorien sind zwar die reinen Formen begrifflicher Urteile gegeben; methodische Überlegungen zur allgemeinen wissenschaftlichen Begriffsfindung sind jedoch nicht Teil des Anliegens seiner Kritik. Diese Leerstelle könnte durch die methodischen Überlegungen Husserls zur Phänomenologie ergänzt werden. Zweitens, und dies lässt auf eine logisch konsistente Möglichkeit der Verknüpfung für die hier verfolgten Zwecke hoffen, sieht Husserl bei den Arbeiten René Descartes, John Lockes, David Humes und insbesondere Immanuel Kants bereits richtungsweisende Überlegungen für die Phänomenologie:

> „So begreift es sich, daß die Phänomenologie gleichsam die geheime Sehnsucht der ganzen neuzeitlichen Philosophie ist. Zu ihr drängt es schon in der wunderbar tiefsinnigen Cartesianischen Fundamentalbetrachtung hin; dann wieder im Psychologismus der Lockeschen Schule, Hume betritt fast schon ihre Domäne, aber mit geblendeten Augen. Und erst recht erschaut sie Kant, dessen größte Intuitionen uns erst ganz verständlich werden, wenn wir uns das Eigentümliche des phänomenologischen Gebietes zur vollbewußten Klarheit erarbeitet haben.“[6]

Im zweiten Teil des Kapitels wird dargelegt, warum zur Analyse von Phänomenen im Zusammenhang mit Computern gerade dieser Programmbegriff vorgeschlagen wird. Dabei werden die Vorteile dieses Begriffs gegenüber anderen möglichen Begriffen wie etwa Software, Code oder Anwendung herausgearbeitet. In diesem Zusammenhang wird auch deutlich, warum einfachere Programmbegriffe für die eingangs beschriebenen Zwecke ungeeignet sind.

6 Husserl, Edmund: Ideen zu einer reinen Phänomenologie und phänomenologischen Philosophie. Niemeyer, 1913. S. 118.

2.1 Erkenntniskritische Grundlagen

2.1.1 Zugang über Immanuel Kants Kritik der reinen Vernunft

Immanuel Kant entwickelt in seiner *Kritik der reinen Vernunft* eine Erkenntnistheorie in der Auseinandersetzung mit den zeitgenössischen philosophischen Strömungen des Rationalismus und des Empirismus. Zu Erkenntnis kann der Mensch nach seinen Ausführungen nur durch ein Zusammenwirken von Anschauungen und Begriffen gelangen. Anschauungen werden dabei durch die Sinnlichkeit des Menschen gegeben, Begriffe entspringen dem Verstand[7]. Weiter unterscheidet Kant zwischen reinen und empirischen Anschauungen und Begriffen:

> „Unsere Erkenntnis entspringt aus zwei Grundquellen des Gemüts, deren die erste ist, die Vorstellungen zu empfangen (die Rezeptivität der Eindrücke), die zweite das Vermögen, durch diese Vorstellungen einen Gegenstand zu erkennen (Spontaneität der Begriffe); durch die erstere wird uns ein Gegenstand g e g e b e n , durch die zweite wird dieser im Verhältnis auf jene Vorstellung (als bloße Bestimmung des Gemüts) g e d a c h t . Anschauung und Begriffe machen also die Elemente aller unsrer Erkenntnis aus, so daß weder Begriffe, ohne ihnen auf einige Art korrespondierende Anschauung, noch Anschauung ohne Begriffe, ein Erkenntnis abgeben *können*. Beide sind entweder rein, oder empirisch. E m p i r i s c h , wenn Empfindung (die die wirkliche Gegenwart des Gegenstandes voraussetzt) darin enthalten ist: r e i n aber, wenn der Vorstellung keine Empfindung beigemischt ist. Man kann die letztere die Materie der sinnlichen Erkenntnis nennen. Daher enthält reine Anschauung lediglich die Form, unter welcher etwas angeschaut wird, und reiner Begriff allein die Form des Denkens eines Gegenstandes überhaupt. Nur allein reine Anschauungen oder Begriffe sind a priori möglich, empirische nur a posteriori.“[8]

Für das Vorhaben der Begriffsbestimmung ist zunächst maßgeblich, dass sich der Begriff nicht auf ein *Ding an sich* beziehen kann. Zugänglich werden uns Gegenstände erst dadurch, dass wir sie durch Sinne erfassen können. Nach Kant setzt jede menschliche Erkenntnis notwendigerweise die menschlichen Sinne voraus. Diese Arbeit folgt Kants Ausführungen insofern, dass ontologische Fragen zu wesentlichen Merkmalen, die Computerprogrammen unabhängig von menschlicher Erkenntnisfähigkeit zukommen, hier nicht weiter verfolgt werden.

Dem Menschen zugänglich sind dagegen Erscheinungen, welche die Gegenstände empirischer Anschauung sind.[9] Auf die Ausgangsfrage dieser Arbeit bezogen bedeutet dies, dass der erste Zugang zu einem Begriff des Programms nach Kant durch die notwendigen Bedingungen aller Erfahrung, die reinen Formen der Anschauung Raum und Zeit, strukturiert ist. Alle direkte Wahrnehmung von Programmen und auch deren Wirkungen kann zu

7 Kant, Immanuel: Kritik der reinen Vernunft. Suhrkamp, 1974. Erstauflage 1781. S. 69. (B 33)

8 Ebda. S. S. 97 (B 74f)

9 Ebda. S. 69 (B 34)

bestimmten Zeiten bestimmten Orten zugeschrieben werden. Computer, Telefone, Datenträger, Kabel, Chips etc. sind greifbar und sichtbar. Programmgesteuerte Gegenstände, vom Industrieroboter über den Kaffeevollautomaten bis zu autonomen Fahrzeugen im Straßenverkehr und autarken Fahrzeugen auf dem Mars, sind in ihren Erscheinungen räumlich und zeitlich. Auf Monitoren werden über verschiedene Techniken Zeichen und Bilder dargestellt, viele Geräte sind hörbar (sei es durch eine gewollte Tonausgabe oder durch Lüftergeräusche), über Braillezeilen werden ertastbare Zeichen ausgegeben. In alltäglichen Verwendungen des Wortes „Programm“ werden diese mitunter über die physischen Gegenstände verortet: Ein Programm läuft auf einem PC, ein Programm ist auf einem Datenträger gespeichert. Nur durch eine solche Verortung, also der Vorstellung von Programmen als physischen Gegenständen, wäre es möglich, Programme zu kopieren oder zu löschen. Diese Handlungen beziehen sich nämlich gerade auf eine Speicherung – sei es in volatilem dynamischen RAM oder auf Lochkarten. Auch technisch vermittelte Erscheinungen, wie z.B. die Zeichen und Bilder, die von einem Monitor dargestellt werden können, sind in Raum und Zeit verortet. Sie können als physisch aufgefasst werden, auch sie sind direkt Objekte empirischer Anschauung.

Die Erscheinungen von Zeichen und Bildern sind für das Vorhaben dieser Arbeit von besonderer Relevanz. Zunächst fallen hierunter die angezeigten Elemente von Benutzeroberflächen, sowohl grafische als auch rein textliche Darstellungen. Kleine Symbole, die in einem Fenstersystem Dateien und Anwendungsprogramme darstellen, Buttons, deren Verhalten nicht nur namentlich Bedienungselementen physischer Maschinen nachempfunden ist und Diagramme, die z.B. den Kontrollfluss eines Programms darstellen, gehören dazu. Ihnen ist gemein, dass sie immer auch für etwas anderes stehen. Die Symbole für Anwendungsprogramme lassen eine bestimmte Funktionalität erwarten – Buttons stehen für Dinge, die passieren, wenn sie geklickt werden. Das Folgenreichste, was uns im Zusammenhang mit Programmen als Zeichen erscheint, ist Sprache. Für die Ausgangsfrage ist natürlich die Sprache, in der uns Programme dargestellt werden, in denen sie geschrieben werden – also Programmiersprache – entscheidend. Die Wahrnehmung von Sprache hat im Zusammenhang mit Programmen jedoch viele weitere Funktionen der Informationsübermittlung. Dies fängt an bei Dateinamen und Verzeichnisstrukturen, die sprachlich ausgedrückt werden, geht über Fehlermeldungen, Stack Traces[10], User Stories[11], Lasten- und Pflichtenhefte bis hin zu E-Mails, Allgemeinen Geschäftsbedingungen und Lizenzdateien. Die Aufzählung ließe sich fortführen; an dieser Stelle soll festgehalten werden, dass Sprache nicht nur als Programmiersprache für Programme relevant ist. Aufgrund der Bedeutung, die Sprache für Computer und Programme hat, lassen sich auch Techniken der Verwendung von Sprache identifizieren, die im Umfeld von Programmen eine wichtige Rolle spielen. Versionsverwaltungssysteme z.B. erlauben einen strukturierten Umgang mit sprachlichen Objekten, die über viele Versionen hinweg geändert werden.

Die sprachlichen Erscheinungen sind dabei in der direkten Wahrnehmung weiterhin an Raum und Zeit gebunden – z.B. erscheint ein Buchstabe zu einer bestimmten Zeit an einem bestimmten Ort auf einem Monitor. Eine Abfolge von solchen Erscheinungen kann im Wahrnehmungsprozess in Zusammenhang gebracht werden. Als Beispiel kann hier ein einfaches Textverarbeitungsprogramm dienen: Ist der bearbeitete Text zu lang, als dass er in Gänze sinnvoll auf dem Monitor dargestellt werden kann, so wird üblicherweise ein Ausschnitt des Textes angezeigt. Durch Verwendung der Tastatur, eines Mausrads oder grafischer Bedienelemente „scrollt“ man durch den Text. Die Funktionalität ist (auch

10 Eine strukturierte Form der Darstellung bestimmter Speicherinhalte, die insbesondere zur Fehlersuche verwendet wird

11 Eine Technik, um gewünschte Programmfunktionalität in kurzen „Geschichten“ darzustellen

namentlich) einer Schriftrolle nachempfunden. Die Veränderung über die Zeit auf dem Monitor, welche Zeichen wo angezeigt werden, erscheint als Bewegung.

Neben physischen und sprachlichen Erscheinungen spielen in dieser Arbeit solche dynamischen Erscheinungen eine wichtige Rolle. Die Ausführung eines Programms erscheint immer als Dynamik, als eine Veränderung über die Zeit. Erst durch die Dynamik kann ein Rechner tatsächlich rechnen. Bei Anwendungen gibt es auch Techniken, um Dynamik bildlich darzustellen: So sollen Fortschrittsbalken einen sonst nicht sichtbaren Vorgang über die Zeit anzeigen, bei manchen Oberflächen erscheint der Mauszeiger als Sanduhr. In beiden Fällen wird räumliche Bewegung vermittelt – Sand der nach unten rieselt oder ein Balken der sich von links nach rechts schiebt. Maus und Mauszeiger sind Techniken, um physikalische Bewegung zunächst in Änderungen von Zahlen zu übersetzen und anschließend wieder als räumliche Bewegung auf dem Monitor darzustellen. Im Zusammenhang mit Computern erscheinen eine Vielzahl weiterer Dynamiken: Ein programmgesteuerter Roboterarm übersetzt Zahlen über Elektromotoren in physische Bewegung. Magnetspeicherplatten werden unter Verwendung von Rotationsbewegungen gelesen. Auch die Herstellung von Software kann als Dynamik begriffen werden: Softwareprojekte sollen in einer bestimmten Zeit ein bestimmtes Ergebnis erreichen, durch aufsteigende Versionierung wird die Entwicklung über die Zeit festgehalten. Compiler überführen menschenlesbare Programmiersprache in ausführbare Maschinensprache in einer festgelegten zeitlichen Abfolge von Einzelschritten, Kommunikationsprotokolle legen fest, wann Nachrichten wie ausgetauscht werden.

Die vorgestellten Erscheinungen sind mit Computerprogrammen sicherlich eng verknüpft. Das Ziel dieser Arbeit – eine Begriffsbestimmung – muss Hinweise darauf geben, welche der Vorstellungen unter den Begriff fallen. Die Anwendung des Begriffs findet nach Kant in Urteilen statt:

> „Alle Urteile sind demnach Funktionen der Einheit unter unsern Vorstellungen, da nämlich statt einer unmittelbaren Vorstellung eine höhere, die diese und mehrere unter sich begreift, zur Erkenntnis des Gegenstandes gebraucht, und viel mögliche Erkenntnisse dadurch in einer zusammengezogen werden. Wir können aber alle Handlungen des Verstandes auf Urteile zurückführen, so daß der V e r s t a n d überhaupt als ein V e r m ö g e n z u u r t e i l e n vorgestellt werden kann. Denn er ist nach dem obigen ein Vermögen zu denken. Denken ist das Erkenntnis durch Begriffe. Begriffe aber beziehen sich, als Prädikate möglicher Urteile, auf irgendeine Vorstellung von einem noch unbestimmten Gegenstande. So bedeutet der Begriff des Körpers etwas, z. B. Metall, was durch jenen Begriff erkannt werden kann. Er ist also nur dadurch Begriff, daß unter ihm andere Vorstellungen enthalten sind, vermittelst deren er sich auf Gegenstände beziehen kann. *Es* ist also das Prädikat zu einem möglichen Urteile, z. B. ein jedes Metall ist ein Körper.“[12]

Wie oben beschrieben spielt die Wahrnehmung von Zeichen und Bildern, insbesondere von Sprache, eine wichtige Rolle bei der Auseinandersetzung mit Computerprogrammen. Da Zeichen für etwas anderes stehen, muss für das weitere Vorgehen die Art von Gegenständen, die unter den Begriff Programm subsumiert werden, genauer untersucht werden. Möglicherweise stellt sich heraus, dass das Relevante, das der Begriff erfassen soll, nicht die Zeichen selbst sind; sondern eben das, wofür sie stehen: die Bedeutung. Die Bedeutung von Text, der in einer Programmiersprache formuliert wird, kann mathematisch sein: als formale Semantik. Im Gegensatz zu einem aus den aufgezählten empirischen Erscheinungen

12 Kant, Immanuel: Kritik der reinen Vernunft. Suhrkamp, 1974. Erstauflage 1781. S. 110. (B 94)

abgeleiteten Begriffs wäre ein Programmbegriff, der sich ausschließlich auf die formale Semantik bezieht, kein empirischer Begriff: Da eine formale Semantik von Programmiersprachen mathematisch gefasst ist können Überlegungen Kants zur Mathematik übertragen werden.

Kant geht es bei der Behandlung der Mathematik unter anderem darum, aufzuzeigen, dass synthetische Urteile a priori möglich sind. Analytische Urteile sind grundsätzlich apriorisch. Er konstatiert, dass mathematischen Urteile insgesamt sowohl apriorisch als auch synthetisch sind[13]. Apriorisch sind mathematische Urteile bei Kant, weil sie notwendigerweise gelten, ohne diese Geltung auf empirische Erkenntnis zu stützen. Synthetisch sind sie, weil sie über eine bloße Zergliederung der Begriffe hinausgehen. Als Beispiel nennt Kant die Summe zweier Zahlen, die ohne Zuhilfenahme von Anschauungen wie den Fingern der Hand nicht berechnet werden könne[14].

Auf Kants zeitgenössische Mathematik bezogen treffen diese Überlegungen zu. Spätestens am Anfang des 20. Jahrhunderts jedoch hat die Grundlagenkrise der Mathematik zu erheblichen Umwälzungen in diesem Bereich geführt. Zunächst versuchte Gottlob Frege in direkter Kritik an Kant die Arithmetik in der Logik zu begründen. Diese logizistische Position übernahm die spätere Arbeit von Bertrand Russell und Alfred North Whitehead. Der heute verbreitet verwendete Formalismus auf Basis der Mengenlehre geht auf David Hilbert, Ernst Zermelo und Abraham Fraenkel zurück[15]. Das ursprüngliche Hilbertprogramm, die Widerspruchsfreiheit der Axiome innerhalb der einzelnen Teilgebiete der Mathematik zu beweisen, kann erwiesenermaßen – aufgrund der Erkenntnis aus Kurt Gödels zweitem Unvollständigkeitssatz – nicht vollendet werden[16]. Erst in der zweiten Hälfte der dreißiger Jahre wurde die Rückführung der gesamten Mathematik auf die Mengenlehre zum Standard etabliert[17]. Die Festlegung des Axiomensystems löst das Problem der unbeweisbaren Widerspruchsfreiheit nicht. Für diese Arbeit ist jedoch relevant, dass überhaupt mit einem solchen Axiomensystem gearbeitet wird, auf dessen Grundlage Aussagen analytisch bewiesen werden können. Bertrand Russell beschreibt mathematische Erkenntnisse später als weder empirisch noch apriorisch, sondern rein verbal[18]. Im Ergebnis kann die Mathematik heute als eine analytische begriffen werden, die weder auf synthetische noch auf empirische Urteile für ihren Erkenntnisfortschritt angewiesen ist. Rudolf Carnap fasst 1935 Logik und Mathematik (aus seinem logizistischen Standpunkt Logik einschließlich der Mathematik) zu Formalwissenschaft zusammen[19]. Sie enthalten analytische Sätze, also solche, die aus der Leeren Menge an Sätzen folgen. Die theoretische Informatik kann analog zur Mathematik als Formalwissenschaft eingeordnet werden.

Auf Computerprogramme bezogen lässt sich übertragen, dass auch die Aussagen über die mathematische Bedeutung, also die formale Semantik, von in Programmiersprache

13 Kant, Immanuel: Kritik der reinen Vernunft. Suhrkamp, 1974. Erstauflage 1781. S. 55 (B 14ff).

14 Ebda. S. 56f (B 15f).

15 Eine umfassende Nachzeichnung der Ereignisse findet sich in: Mancosu, P., Zach, R., & Badesa, C.: The development of mathematical logic from Russell to Tarski, 1900-1935. 2008. in: Haaparanta, L.: The development of modern logic. Oxford University Press, 2009. S. 318-470.

16 Ebda. S. 403.

17 Ebda. S. 352.

18 Russell, Bertrand, et al.: Philosophie des Abendlandes: ihr Zusammenhang mit der politischen und der sozialen Entwicklung. Europaverlag, 1950. S. 839.

19 Carnap, R.: Formalwissenschaft und Realwissenschaft. In: Erkenntnis, Vol. 5 Issue 1. 1935. S. 30-37.

verfasstem Text analytisch sind. Bezieht sich der Programmbegriff also ausschließlich auf diese Bedeutung, so ist er unabhängig von aller empirischen Erkenntnis:

> „Es kommt hier auf ein Merkmal an, woran wir sicher ein reines Erkenntnis von empirischen unterscheiden können. Erfahrung lehrt uns zwar, daß etwas so oder so beschaffen sei, aber nicht, daß es nicht anders sein könne. Findet sich also e r s t l i c h ein Satz, der zugleich mit seiner N o t w e n d i g k e i t gedacht wird, so ist er ein Urteil a priori; ist er überdem auch von keinem abgeleitet, als der selbst wiederum als ein notwendiger Satz gültig ist, so ist er schlechterdings a priori.“[20]

Aussagen über Programme, die mathematisch bewiesen sind, sind notwendig gültig. Diese Eigenschaften wiederum beziehen sich jedoch ausschließlich auf Programme als mathematische Objekte. Die definierten Objekte der Mathematik sind für Kant weder *rein* noch *empirisch*, sondern *willkürlich gedacht*:

> "Da also weder empirisch, noch a priori gegebene Begriffe definiert werden können, so bleiben keine andere als willkürlich gedachte übrig, an denen man dieses Kunststück versuchen kann.“[21]

Ein entsprechender Programmbegriff würde alle genannten empirischen Erscheinungen ausschließen. Auch die Menge an Zeichen, in der ein Programm geschrieben ist, wäre nicht selbst Programm. Lediglich die formale Semantik, für die diese Zeichen stehen, ist das Programm. Nur Programme in diesem mathematischen Sinne lassen sich formal verifizieren: Die Beweise, die im Zuge der Programmverifikation geführt werden, beziehen sich auf das mathematische Objekt. Sie könnten sich auch gar nicht auf die physischen Gegenstände empirischer Erscheinungen beziehen. Als physische Gegenstände haben sie grundsätzlich keine mathematisch beweisbaren Eigenschaften. Zur Illustration: Selbst wenn die formale Semantik des Quellcodes eines Anwendungsprogramms dahingehend verifiziert wurde, dass das Ergebnis einer enthaltenen Berechnung korrekt ist, kann der Strom ausfallen.

Ein mathematischer Programmbegriff hat seine Berechtigung innerhalb der formalen Grundlagen der Informatik. Damit lassen sich die Objekte erfassen, deren Eigenschaften in Verifikationen bewiesen werden können – in diesem Rahmen können z.B. auch Aussagen über die Komplexität von Berechnungen oder die generelle Berechenbarkeit von Problemen formuliert werden. Für eine Erörterung menschlichen Handelns, welches sich auf Programme bezieht, oder theoretische Überlegungen zum wechselseitigen Verhältnis von Technik und Gesellschaft greift der Begriff jedoch zu kurz. Programme, die darunter fallen, können weder laufen noch abstürzen. Sie tun nichts. Sie steuern keine Roboter, führen keine Berechnungen (im Sinne einer Dynamik) durch, sie haben in einem praktischen Sinne keine Ein- und Ausgabe. Mathematische Programme wirken erst durch einen menschlichen Bezug auf sie: z. B. wenn eine Erkenntnis über die Rechenkomplexität eines bestimmten Algorithmus Menschen dazu veranlasst, eine andere Lösung zu suchen. In diesem Sinne sind mathematische Programme auch für die hier durchgeführte Begriffsbestimmung relevant – sie erschöpfen jedoch nicht alle Gegenstände, die unter diesen Begriff fallen müssen. Aus diesem Grund kann der hier entwickelte Begriff weder *rein* noch *willkürlich gedacht* sein, er muss auch *empirische* Gegenstände subsumieren – er erfasst daher das Computerprogramm als Erfahrungsgegenstand.

20 Kant, Immanuel: Kritik der reinen Vernunft. Suhrkamp, 1974. Erstauflage 1781. S. 46 (B 4)

21 Ebda. S. 624 (B 757)

2.1.2 Zugang mit Edmund Husserls Phänomenologie

Edmund Husserl unternimmt in seinem Werk *Ideen zu einer reinen Phänomenologie und phänomenologischen Philosophie*[22] den Versuch, die Philosophie und die Naturwissenschaften auf eine phänomenologische Grundlage zu stellen. Ausgangspunkt aller Erkenntnis sind dabei die Bewusstseinsinhalte. Ziel ist es, Vorurteile und falsche Annahmen dadurch zu überwinden, dass die der menschlichen Wahrnehmung zugänglichen Dinge allein auf die direkt erfahrbaren Wahrnehmungsakte beschränkt werden. Diese Wahrnehmungsakte sind dabei intentional: Bewusstsein ist immer Bewusstsein von etwas.

> „Empirische Anschauung, speziell Erfahrung, ist Bewußtsein von einem individuellen Gegenstand, und als anschauendes »bringt sie Ihn zur Gegebenheit«, als Wahrnehmung zu originärer Gegebenheit, zum Bewußtsein, den Gegenstand »originär«, in seiner »l e i b h a f t i g e n « Selbstheit zu erfassen.“[23]

Husserl greift dabei auf Franz Brentanos Aktpsychologie zurück, insbesondere übernimmt er für sein Vorhaben den Begriff der Intentionalität des Bewusstseins[24]. Diese Intentionalität ist die Beziehung auf Gegenstände, die eine als Akt begriffene Wahrnehmung notwendigerweise vollzieht: Wahrnehmung ist auch immer Wahrnehmung von etwas. Wahrnehmung ist dabei nur eines von vielen intentionalen Erlebnissen, weitere Beispiele sind Meinung, Erinnerung, Urteil etc[25]. Diese Fassung der Phänomene als Gegenstände intentionaler Wahrnehmungstätigkeit ermöglicht Husserl, ontologische Untersuchungen über das Wesen des Bewusstseins und der Gegenstände, auf die es intentional gerichtet ist, vorzunehmen. Husserl grenzt die Phänomenologie dabei scharf von der empirischen Psychologie ab, da sie als Wesenswissenschaft den Tatsachenwissenschaften Psychologie, Naturwissenschaften, Geisteswissenschaften und Sozialwissenschaften gegenüber stehe[26].

Für das Vorhaben dieser Arbeit sind dabei insbesondere die methodischen Überlegungen zur Phänomenologie relevant. Sie ermöglichen, die in Wahrnehmungsakten konstituierten Formen von Computerprogrammen zu sammeln, nach ihren Eigenschaften zu sortieren und schließlich durch die Reduktion auf wesentliche Aspekte zum Begriff zu bringen. Ausgangspunkt für dieses Vorgehen sind dabei die unter Bezug auf Kant identifizierten Erscheinungen.

Die Anwendung der Phänomenologie beschränkt sich hier jedoch nicht allein auf Wahrnehmungsakte. Auch andere auf Computerprogramme bezogene Handlungen können ihrem intentionalen Charakter nach untersucht werden. Die Frage, der dabei im Hinblick auf die Begriffsbestimmung nachgegangen werden soll ist: Welcher Art muss der Gegenstand eines bestimmten intentionalen Erlebnisses sein? Z.B. unterstellt die Handlung, ein Programm zu lesen, dass es lesbar ist. Damit ist es notwendigerweise Schrift, also Sprache. Diese Charakterisierung gilt jedoch nur unter einer Begriffsbedeutung, unter der das Lesen eine prinzipiell denkbare Handlung darstellt.

22 Husserl, Edmund: Ideen zu einer reinen Phänomenologie und phänomenologischen Philosophie. Niemeyer, 1913.

23 Ebda. S. 11

24 Siehe ebda. S. 172ff

25 Ebda. S. 182

26 Ebda. S. 2ff.

Zunächst muss darauf hingewiesen werden, dass die phänomenologischen Methoden hier in gewisser Weise zweckentfremdet werden. Diese Arbeit entwickelt anhand der Phänomene, auf die sich (Wahrnehmungs-)Handlungen beziehen, einen empirischen Begriff. Dieser ist nicht eidetisch, kein Wesensbegriff, und damit kein Begriff der reinen Phänomenologie. Die Methoden zur Analyse der Gegenstände, auf die sich das Bewusstsein intentional bezieht, können trotzdem für das Vorhaben fruchtbar gemacht werden.

Die phänomenologischen Methoden, die für diese Arbeit von Relevanz sind, sind die genaue Deskription, Reflexion[27] und die Enthaltung (Epoché) zur Reduktion. Die genaue Beschreibung von Wahrnehmungsprozessen ist bei Husserl dabei die Methode, um wesenhafte Erkenntnisse über die Wahrnehmung im Allgemeinen zu erlangen[28]. Analog wird dies für andere Formen intentionalen Erlebens durchgeführt. Die Reflexion ist bei Husserl zentral für Erkenntnisse des Bewusstseins über sich selbst. Reflexion wird dabei als Bewusstseinsmodifikation verstanden, mit der ein vorgegebenes Erlebnis umgewandelt wird[29]. Die phänomenologischen Reduktion, das „Absehen von der ganzen Welt“[30], ist bei Husserl die Grundvoraussetzung gesicherter phänomenologischer Erkenntnis. Die Reduktion geschieht dabei durch Epoché, die Enthaltung aller Vorannahmen, inklusive aller wissenschaftlichen Erkenntnisse. Diese ist notwendig für Husserls Vorhaben, die Gesamtheit der Wissenschaften auf ein phänomenologisches Fundament zu stellen. Später beschreibt er diese Reduktion so:

> „Offenbar ist allem voran erfordert die Epoché hinsichtlich aller objektiven Wissenschaften. Das meint nicht bloß eine Abstraktion von ihnen, etwa in der Art eines fingierenden Umdenkens des gegenwärtigen menschlichen Daseins, als ob darin nichts von Wissenschaft vorkäme. Vielmehr gemeint ist eine Epoché von jedem Mitvollzug der Erkenntnisse der objektiven Wissenschaften, Epoché von jeder kritischen, an ihrer Wahrheit oder Falschheit interessierten Stellungnahme, selbst zu ihrer leitenden Idee einer objektiven Welterkenntnis. Kurzum, wir vollziehen eine Epoché hinsichtlich der ganzen objektiven theoretischen Interessen, der gesamten Bezweckungen und Handlungen, die uns als objektiven Wissenschaftlern oder auch nur als Wißbegierigen eigen sind.“[31]

In zwei wesentlichen Aspekten weicht das Vorgehen in dieser Arbeit von Husserls phänomenologischer Methode ab. Erstens ist das Ziel, einen fruchtbaren Begriff für Computerprogramme zu entwickeln. Wie bereits im ersten Abschnitt des Kapitels ausgedrückt wird dabei kein Versuch einer ontologischen Bestimmung unternommen. Der gesuchte Begriff wird daher deskriptiv und nicht eidetisch, es ist nicht das Ziel, Wahrheit über *das* Programm oder das Wesen *des* Programms zu erfassen. Vielmehr soll der Begriff durch eine Zuordnung und Sortierung empirischer Erkenntnis fruchtbar für Überlegungen zum Programm als Technik sein. Nach Husserl wäre das Vorgehen also nicht phänomenologisch, sondern im Bereich der Tatsachenwissenschaften anzusiedeln. Für den Zweck der Begriffsbestimmung ist das hinreichend, der Programmbegriff hat keinen Anspruch auf transzendentalphilosophische Erkenntnis. Gleichwohl erfolgt die Analyse der Bedeutungen, in denen Programme gefasst werden, unter Rückbezug auf die Phänomene, in denen sie sich intentionalem Handeln

27 Siehe Husserl, Edmund: Ideen zu einer reinen Phänomenologie und phänomenologischen Philosophie. Niemeyer, 1913. S. 147ff.

28 Ebda. S. 140.

29 Ebda. S. 148.

30 Ebda. S. 95. Zu Epoché und phänomenologischer Reduktion siehe ebda. S. 56f, 94f und 108ff.

31 Husserl, Edmund: Husserliana 6, Die Krisis der Europaischen Wissenschafien und die Transzendentale Phänomenologie, herausgegeben von Walter Biemer. Nijhoff, 1976. S. 138f.

darbieten.

Zweitens erfolgt die Epoché hier lediglich auf Vorannahmen über den begrifflich zu erfassenden Gegenstand, also Computerprogramme. Vorwissen über die grundsätzliche Funktionalität von Computern, Softwareentwicklungsprozesse und Formalsprachen sind notwendig, um die Phänomene in Bezug zu anderen Begriffen der Soziologie und Philosophie zu setzen. Die Urteilsenthaltung bezieht sich jedoch neben existierenden Programmbegriffen auch auf deren erfahrungsunabhängige Existenz, subjektive Zuschreibungen und sehr begrenzt anwendbare Charakteristiken (ein Beispiel für eine solche begrenzt anwendbare Charakteristik für Programme wäre Intelligenz: Während auf manche Techniken im Bereich maschinellen Lernens bezogen eine Diskussion über die Intelligenz von Programmen fruchtbar sein kann ist sie kein sinnvolles Kriterium für einen allgemeinen Programmbegriff).

Zur Illustration wird an dieser Stelle die Anwendung der skizzierten Methoden an einem Beispiel vollzogen und dem Verständnis unter einem einfachen Programmbegriff gegenüber gestellt. Als Beispiel dient wieder das Lesen eines Programms. Als einfacher Begriff wird die Definition von Programm der ISO verwendet: Ein Programm ist danach:

> „[a] syntactic unit that conforms to the rules of a particular programming language and that is composed of declarations and statements or instructions needed to solve a certain function, task, or problem“[32]

Das Lesen würde also ein Erfassen der Syntax beinhalten und über die Programmiersprache auch einen Bezug zur Semantik herstellen. Den Zweck des Programms zu erfassen würde über das Lesen bereits hinausgehen.

Das hier vorgeschlagene methodische Vorgehen enthält sich zunächst vorab aller Urteile, was der Gegenstand des Lesens sei (also z.B. der zitierten Definition). Weiter müsste der Vorgang *Lesen* für eine genaue Deskription konkretisiert werden. Dafür wird empirisch ein beliebiges Beispiel einer solche Situation beobachtet: Ein Mensch sitzt vor einem Laptop. Er schließt einen USB-Speicher an. Mithilfe einer Entwicklungsumgebung öffnet er durch Bewegen einer Maus und Tastatureingabe eine Datei auf diesem Speicher. Anschließend werden auf dem Monitor Zeichen angezeigt, die verschieden eingefärbt sind. Durch Eingabe verändert er Stück für Stück die angezeigten Zeichen. Er deutet die Zeichen in einer ihm bekannten Sprache und nutzt die Einfärbung zur schnelleren Orientierung. Er leitet daraus eine Bedeutung der Gesamtheit der erfassten Zeichen ab.

In einem nächsten Schritt der Reflexion können nun Aspekte dieser Beschreibung identifiziert werden, die notwendig für den Vorgang „Lesen“ sind. Der USB-Speicher ist z.B. gegen andere Datenträger austauschbar, die Entwicklungsumgebung gegen andere Möglichkeiten der Darstellung, auch gegen reine Textverarbeitung. Damit ist die Einfärbung nicht essenziell. Der Laptop könnte gegen andere Computer oder sogar gegen Papier getauscht werden, ohne dass die Handlung „Lesen“ unmöglich wird. Der Monitor könnte gegen eine Braillezeile getauscht werden. Damit ist auch das optische Erfassen der Zeichen nicht maßgeblich.

Die notwendigen Invarianten sind dann die Voraussetzungen für die Lesbarkeit eines Programms. Diese sind hier: Zeichen, die für menschliche Sinne wahrnehmbar gemacht werden, eine erlernbare Sprache und eine Bedeutung der Zeichen in dieser Sprache. Gegenüber der Anwendung des einfachen Begriffs ergeben sich zwei Vorteile: Erstens muss nicht auf Begriffe bestimmter Programmierparadigmen zurückgegriffen werden

32 International Organization for Standardization. (2015): Information technology - Vocabulary. (ISO/IEC Standard 2382:2015(en)). Siehe https://www.iso.org/obp/ui/#iso:std:iso-iec:2382:ed-1:v1:en (abgerufen am 29.06.2018).

(Deklarationen, Statements, Instruktionen). Zweitens lässt sich durch dieses Vorgehen die Relevanz bestimmter Eigenschaften der Handlungen für unser Verständnis von Computerprogrammen verdeutlichen, hier z.B. die Techniken der Übertragung von Zuständen (des USB-Speichers) in für Menschen wahrnehmbare Phänomene und die prinzipielle Erlernbarkeit der Sprache. Die ISO-Definition muss dies nicht leisten, da sie Programme nicht als lesbar erfasst. Auch wenn nicht jedes Programm notwendigerweise menschenlesbar ist, so kommt doch der Möglichkeit des Lesens eine solch zentrale Bedeutung für menschliche Bezüge auf Programme zu, dass ein für Reflektionen über Technik ausreichend komplexer Begriff diese Eigenschaft aufgreifen muss.

Mithilfe dieser Zugänge können an dieser Stelle mehrere Klassen von Phänomenen identifiziert werden. Im Hauptteil der Arbeit werden aus diesen Klassen die Bedeutungsdimensionen des Programmbegriffs aufgebaut. Die Erfassung der Klassen erfolgt dabei ausgehend von den im vorherigen Abschnitt bereits unterschiedenen Arten von Erscheinungen, die im Zusammenhang mit Computern und möglichen Programmen ausgemacht wurden.

Den Erscheinungen, die unmittelbar sinnlich gegeben sind, entsprechen die physischen Phänomene. Diese umfassen alle Gegenstände intentionaler Akte, die als räumlich und zeitlich wahrgenommen, gedacht, konzipiert, hergestellt etc. werden. Von Datenträgern über verschiedene Techniken der Ein- und Ausgabe, Tönen, durch Monitore produzierten Lichtwellen bis hin zu Kabeln, integrierten Schaltkreisen, Platinen, Lochkarten, bedrucktem Papier und digitalisierten Fabriken sind dies Phänomene, die sich in einem bestimmten Raum in einer bestimmten Zeit darstellen, also als räumlich und zeitlich intendiert werden.

Neben den als physisch wahrgenommenen Gegenständen führt uns eine phänomenologische Betrachtung noch zu solchen, die sich auch als räumlich und/oder zeitlich darstellen, deren Raum und Zeit sich jedoch vom als physisch intendierten radikal unterscheiden. Diese sollen virtuell-physische Gegenstände genannt werden. Zur Illustration kann hier an Fenster einer grafischen Benutzeroberfläche gedacht werden. Diese werden vom Bewusstsein als Gegenstände intendiert. Neben den zwei physischen Dimensionen eines Monitors, auf dem sie verschoben werden, konstruiert sie der Wahrnehmungsakt noch in einer dritten, virtuellen Dimension: Zwei Fenster können als vor- bzw. hintereinander wahrgenommen werden, sich gegenseitig verdecken. Dies geschieht rein virtuell, als physisches Phänomen wird ein verdecktes Fenster durch den Monitor schlicht nicht dargestellt. Analog gilt dies für Dateisysteme, die als räumlich wahrgenommen werden können: Verschiedene Unterverzeichnisse werden in der Wahrnehmung als unterschiedliche Orte intendiert, entsprechend können Dateien „verschoben“ werden. Für Netze gilt dies auch, prominent anschaulich insbesondere im World Wide Web, der englische Begriff *website* drückt hier die Räumlichkeit noch besser aus als der deutsche *Webauftritt*. Eine Datei, die sich über längere Zeit unter der gleichen URL findet, wird als örtlich konstant wahrgenommen, „man weiß, wo man sie findet“, auch wenn der Server, auf dem sie liegt mittlerweile ein anderer, tausende Kilometer entfernter, ist.

Auch zeitliche Virtualisierung spielt eine wichtige Rolle. Versionsnummern werden als vorher und nachher intendiert, das Verstreichen physischer Zeit zwischen verschiedenen Versionsveröffentlichungen wird erst in einem zweiten Schritt nachvollzogen. Das Betrachten von Quellcode einer alten Programmversion lässt eine „Vergangenheit“ betrachten, jedoch in keinem physischen Sinn. Kommunikationsprotokolle legen zeitliche Abfolgen von Nachrichten fest, oft ohne eine physische Zeit, die hier verstreicht, festzulegen. Instruktionen einer imperativen Programmiersprache werden als zeitliche Abfolge intendiert – ohne bei gleich bleibender Semantik auch nur eine gleiche tatsächliche Reihenfolge der Ausführung zu

garantieren: Ein Compiler kann Instruktionen für bessere Programmperformance umschichten, solange das Verhalten zur Laufzeit sich dadurch funktional nicht ändert – und bei einem üblichen Pipeline-Prozessor kann von einem strikten Nacheinander von Befehlen ohnehin nicht gesprochen werden.

Als nächste Klasse von Phänomenen sollen hier sprachliche Phänomene genannt werden. Diese umfassen alles, was ich als Sprache wahrnehme, verstehe, lese oder schreibe. Wie bereits angedeutet, gibt es zwei Ebenen, in denen sich Bewusstsein auf diese Gegenstände beziehen kann: Die Gegenstände können als reine Zeichen wahrgenommen werden, deren Aussehen, Anordnung, Farbe, Größe usw. auf dieser Ebene die relevanten Merkmale sind. Z.B. ist eine auf diese Ebene bezogene Handlung das Kopieren von Sprache. Wenn die Löcher auf einer Lochkarte als Sprache verstanden werden, so können diese auf eine andere Lochkarte übertragen werden, ohne sich irgendwie auf die Bedeutung zu beziehen. Auch eine Aussage z. B. über die Länge von Quellcode bezieht sich auf diese Syntax. Die zweite sprachliche Ebene neben der Syntax ist die Bedeutung der Zeichen. Die formale Semantik wurde bereits genannt. Zu dieser Ebene sprachlicher Phänomene gehört jedoch auch eine natürlichsprachliche Bedeutung: „while“ ist nicht nur ein Schlüsselwort in vielen Programmiersprachen, sondern auch ein englisches Wort. Für die Lesbarkeit von Quellcode ist es nicht unwichtig, dass die formale Semantik und die natürlichsprachliche Bedeutung zusammenhängen. Natürlichsprachliche Phänomene mit Bezug zu Computern begegnen uns auch in der Form von Dokumentationen, User Stories, Ein- und Ausgabe, rechtlichen Dokumenten, Namen (auch Bezeichnern) und Kommentaren.

Die Bewegungserscheinungen lassen sich zu einer Klasse zusammenfassen, die prozessuale Phänomene heißen sollen. Diese umfassen alles, was als eine Veränderung über einen zeitlichen Horizont intendiert wird. Dazu gehören Gegenstände, die ich als Vorgang meine, plane, erhoffe, durchführe etc. Ein zentrales Beispiel ist hier die Ausführung einer Anwendung: Üblicherweise können Zeitpunkte von Anfang und Ende der Ausführung wahrgenommen werden, dazwischen passiert etwas. Und dieses Etwas ist (zumindest teilweise) der Grund, eine Anwendung überhaupt auszuführen. Weitere Beispiele aus dem Umfeld der Programmierung sind Softwareentwicklungsprozesse, im weitesten Sinne Sozialtechniken, mit deren Hilfe Software hergestellt wird. Quellcodeübersetzungen sind eine besondere Form der Anwendungsausführung, Compiler übersetzen hier Code in eine andere Programmiersprache oder in direkt ausführbaren Maschinencode. Die Herstellung von Software bezieht sich dabei immer wieder intentional auf das prozessuale Verhalten zur Laufzeit: eine neue Funktionalität soll eingebaut werden, ein Laufzeitfehler soll vermieden werden, die Performance soll verbessert werden, die Sicherheit erhöht werden usw. Programme weisen im Zusammenhang mit prozessualen Phänomenen eine Besonderheit auf: Ihr Verhalten über die Zeit ist niemals aus dem Programm alleine heraus erklärbar. Ein laufendes Programm wird immer *auf* weiteren Gegenständen ausgeführt: auf einem Prozessor, auf einem eingebetteten System oder auf einem vollständigen PC mit Peripheriegeräten und Netzwerkzugang. Das Laufzeitverhalten eines Programms kann nur als eingebettet in ein System weiterer Gegenstände konstituiert werden. Im weiteren Verlauf der Arbeit stellt sich heraus, dass diese Einbettung so entscheidend ist, dass die aus prozessualen Phänomenen entwickelte Bedeutungsdimension Programme als eingebettete Gegenstände begreift.

Besonders im Zusammenhang mit Laufzeitverhalten begegnet uns noch eine weitere Klasse von Phänomenen, die hier zunächst in weitestem Sinne als gesellschaftliche Phänomene bezeichnet werden sollen: Z.B. kann für die Verwendung eines Computers eine Authentifizierung erforderlich sein. Zur Anwendung muss also per Passwort, Fingerabdruck oder Spracherkennung, nachgewiesen werden, dass das Recht dazu vorliegt. Konventionen

kommen an vielen Stellen zum Ausdruck: Von der Sprache, in der Ein- und Ausgabe stattfinden, über die Darstellung von Fehlermeldungen bis hin zu Erwartungen an grafische Oberflächen. Auch bei der Herstellung von Software zeigen sich gesellschaftliche Phänomene: in Entwicklungsteams gibt es Gruppenprozesse, gegenseitiges Lehr- und Lernverhalten, eigene Konventionen innerhalb der Teams. Softwaretechnik greift die Entwicklung als gesellschaftliches Phänomen auf und zielt auf eine erfolgreiche Steuerung ab. In größerem Maßstab gibt es Konventionen und Trends bezüglich Programmiersprachen und Stil, bis hin zu Überlegungen zu einer *Hackerkultur* und *Hackerethik*. Auch rechtliche Phänomene treten im Zusammenhang mit Computern und Software auf.

An dieser Stelle sei darauf hingewiesen, dass die Klassen von Phänomenen nicht streng unterschieden sind. Ein Softwareentwicklungsprozess stellt sich sowohl prozesshaft als auch als gesellschaftliches Phänomen dar. Code, der auf einem Monitor angezeigt wird, kann intentional als Sprache wahrgenommen werden oder intentional als physisches Phänomen, als Lichtwellen. Die Abgrenzung zwischen den einzelnen Bedeutungsdimensionen des Programmbegriffs und wie sich Bewusstsein letztendlich intentional auf Programme bezieht wird in der eigentlichen Begriffsentwicklung deutlich. Die Bedeutungsdimensionen werden dort aus den hier aufgeführten Klassen von Phänomenen entwickelt – den physischen, virtuell-physischen, sprachlichen, prozessualen und gesellschaftlichen Phänomenen. Dort wird auch herausgearbeitet, dass die gesellschaftlichen Phänomene für den begrifflichen Zusammenhang maßgeblich sind, da der Zusammenhang zwischen den ersten vier Klassen soziotechnisch reproduziert wird.

2.2 Grundlegung eines gesellschaftlich verfassten Programmbegriffs

Aus den bisherigen Ausführungen geht bereits hervor, dass die hier vorgeschlagene Antwort auf die Ausgangsfrage ein Programmbegriff mit mehreren Bedeutungsdimensionen sein muss. Programme sind Gegenstände, die ich als physisch begreifen und darum kopieren oder löschen kann. Gleichzeitig sind sie Gegenstände, die ich als sprachlich begreifen und daher lesen oder schreiben kann. Und sie sind auch als prozessuale Gegenstände, die in ein System eingebettet laufen oder es zum Absturz bringen können, begreifbar. Diese Gegenstandsformen sind als unterschiedliche Perspektiven, die ausdrücken als was ein Programm jeweils intendiert wird, klar unterscheidbar. Gleichzeitig sind ihre wechselseitigen Bezüge zueinander jedoch auch entscheidend für die Bedeutung, die Programmen als Technik insgesamt zukommt.

Der begriffliche Zugang, welcher diese beiden Ansprüche (die Unterscheidung und die Erfassung der wechselseitigen Bezüge) miteinander verbindet, ist in doppeltem Sinne gesellschaftlich verfasst. Erstens muss er in den menschlichen Bezügen zu den einzelnen Bedeutungsdimensionen gesellschaftliche Phänomene ernst nehmen. Wenn z.B. Software in einem Unternehmen oder einem Verein eingesetzt werden soll, wenn Programmierung als Dienstleistung verkauft wird, oder wenn Code unter einer Lizenz für Freie Software entwickelt und veröffentlicht wird, so sind diese Vorgänge nur in den Bezügen zu menschlichen Akteuren erklärbar. Sie können in den allermeisten Fällen als soziales Handeln[33] begriffen werden[34].

Zweitens kann die Verknüpfung zwischen den einzelnen Begriffsdimensionen, die hier entwickelt werden, als gesellschaftliches Phänomen begriffen werden. Die Möglichkeit, Quellcode von einer Datei auf einem *Solid-State-Drive* (SSD), einer als Flashspeicher realisierten Festplatte, zu lesen, setzt eine ganze Reihe unterschiedlicher Techniken voraus, auf die dabei zurückgegriffen wird. Ohne eine Gesellschaft, in der diese Techniken verfügbar sind, ist eine solche Handlung nicht denkbar. Es ist also gesellschaftlich voraussetzungsreich, dass eine Verknüpfung des physischen Gegenstands (dem SSD) mit einem sprachlichen Gegenstand überhaupt stattfinden kann.

Im letzten Kapitel wurde zwischen fünf Klassen von mit Programmen verknüpften Phänomenen unterschieden, den physischen, virtuell-physischen, sprachlichen, prozessualen und gesellschaftlichen Phänomenen. Um zu anwendbaren Begriffsdimensionen zu gelangen, müssen diese Klassen modifiziert werden. Direkt als physisch oder virtuell-physisch intendierte Programme sollen als räumlich-zeitliche Gegenstände aufgefasst werden – beide Gegenstandsformen sind ihrer Konstitution nach durch Raum und Zeit bestimmt, unabhängig davon, ob Raum und Zeit jeweils als physisch oder virtuell konstituiert werden. Sprachlich konstituierte Gegenstände lassen sich danach unterscheiden, ob die Intention sich auf die syntaktisch geordneten Zeichen oder auf ihre Bedeutung bezieht. Damit sind Programme als syntaktische und semantische Gegenstände zwei weitere relevante Bedeutungsdimensionen. Ausgeführte Programme, die als prozessual wahrgenommen werden, sind als Gegenstand durch ihre Einbettung in ein ausführendes System gekennzeichnet. Diese Einbettung ist dabei

33 Siehe Weber, Max: Wirtschaft und Gesellschaft. Mohr Siebeck, 1922. S. 1ff.

34 Der begriffliche Zugang dieser Arbeit wird als *gesellschaftlich* verfasst verstanden. Dies geht über Max Webers Begriff des *Sozialen* (z.B. *sozialen Handelns)* hinaus: Die Veröffentlichung von Code unter einer bestimmten Lizenz ist nicht nur ihrem subjektiven Sinn nach auf das Handeln anderer bezogen, sie ist darüber hinaus nur innerhalb einer Vielzahl stabilisierter sozialer Beziehungen denkbar, die dieser Lizenz ihre Bedeutung geben. Der rechtliche Rahmen innerhalb dessen eine Lizenz bestimmte Handlungen erlaubt oder verbietet ist ein *gesellschaftlicher* (und nicht lediglich sozialer) Rahmen; der hier entwickelte Programmbegriff hat den Anspruch, diese gesellschaftlichen Bezüge aufzugreifen.

für die Bedeutung des Programms zentral: Ohne ein System, welches das Programm ausführt, kann dieses nicht als Prozess begriffen werden. Bei der Einbettung können auch menschliche Bezüge (z.B. zu den Anwender_innen) mit betrachtet werden, Programme sind auch gesellschaftlich eingebettet. Die Bedeutungsdimension des Begriffs bezieht sich damit auf das Programm als eingebetteten Gegenstand – unabhängig davon, ob die Einbettung rein technisch oder auch durch menschliche Interaktion geprägt ist. Die fünfte und letzte Bedeutungsdimension, das Programm als Assoziation, greift diejenigen Phänomene auf, die die anderen vier Dimensionen miteinander verbinden. Diese Verbindung wird als gesellschaftlich verfasst begriffen; so sind z.B. die auf einem Datenträger gespeicherte Programmdatei, der darin enthaltene Quellcode, seine Bedeutung und sein Verhalten zur Ausführung nur in einem bestimmten soziotechnischen Kontext miteinander verknüpft: Durch Schreib- und Lesetechniken, die verwendete Programmiersprache, durch einen Compiler und ein ausführenden System. Zusammengenommen ermöglicht dieser Kontext die Assoziationen zwischen den ersten vier Bedeutungsdimensionen. Damit stellt die gesellschaftliche Verfasstheit der Assoziationen einen zentralen Aspekt des hier entwickelten Programmbegriffs dar.

An dieser Stelle muss eine Grundannahme getroffen werden: Computerprogramme sind, zumindest mittelbar, Produkte menschlicher Handlungen. Das schließt explizit maschinengenerierten Code nicht aus, auch dieser geht mittelbar auf menschliche Handlungen zurück. Erst durch diese Einschränkung sind die Gegenstände grundsätzlich einem sinnverstehenden Bezug zugänglich: Ein Naturereignis, z.B. ein quantenmechanischer Tunneleffekt, ist aus sich selbst heraus sinnlos. Ein Mensch kann diesen Effekt verstehen, kann die Gleichungen kennen, mit denen er in der Physik beschrieben wird. Ohne Bezug zu menschlichem Handeln können dem Effekt jedoch keine Zwecke, keine Werte oder Traditionen zugeschrieben werden. Erst in der Ausnutzung dieses Effekts in einer Technik kommt ihm ein Sinn zu, im Beispiel das Ausnutzen des quantenmechanischen Effekts zur Speicherung von Information auf einem SSD. Das SSD ist also, im Gegensatz zum isolierten physischen Effekt, sinnverstehender Erfahrung zugänglich.

Der sinnverstehende Zugang zu Programmen ist in vielen Fällen ein intersubjektiver: Der Sinn, der verstanden (oder missverstanden) wird, ist der subjektiv gemeinte Sinn[35] eines oder mehrerer Menschen. Es gibt hier auch auf sich selbst bezogen subjektives Sinnverstehen: Wenn z.B. selbstgeschriebener Quellcode reflektiert wird, der Sinn einzelner Teile für den Zweck hinterfragt wird, ist zunächst kein anderes Bewusstsein an den Vorgängen beteiligt. Allein durch zeitlichen Abstand, beispielsweise beim Lesen des eigenen Quellcodes drei oder vier Monate später, können sich aber bereits Parallelen zu Vorgängen des Fremdverstehens ergeben: Der eigene subjektiv gemeinte Sinn kann in der Reflektion durchaus rätselhaft erscheinen. Ein wesentlicher Unterschied zwischen rein subjektiven und intersubjektiven Bezügen liegt in der Konstitution eines anderen Ichs, das an intersubjektiven Vorgängen beteiligt ist. Wenn ich den Code eines anderen Ichs lese, ist mir bewusst, dass dieses Ich eigenes Bewusstsein ist, eigene Zwecke für den Code setzt, ein eigenes Wissen über Programme, Programmiersprachen und intendierte Domäne besitzt, einen eigenen Stil in der Programmierung hat usw. Die Erfahrung des anderen Bewusstseins in intentionalen Bezügen zu Programmen soll nun näher untersucht werden.

Da in Edmund Husserls Phänomenologie alle Erkenntnis im Bewusstsein selbst seinen Ursprung hat, muss jede Form von möglicher Intersubjektivität entsprechend aus diesem Bewusstsein heraus erklärbar sein. Husserl führt für seine Fassung eines transzendentalen Idealismus dafür in seinen *Cartesianischen Meditationen* den Begriff der Appräsentation ein.

35 Vgl. Weber, Max: Wirtschaft und Gesellschaft. Mohr Siebeck, 1922. S. 2f.

„Erfahrung ist Originalbewußtsein, und in der Tat sagen wir im Falle der Erfahrung von einem Menschen allgemein, der Andere stehe selbst *leibhaftig* vor uns da. Anderseits hindert diese Leibhaftigkeit nicht, daß wir ohne weiteres zugestehen, daß dabei eigentlich nicht das andere Ich selbst, nicht seine Erlebnisse, seine Erscheinungen selbst, nichts von dem, was seinem Eigenwesen selbst angehört, zu ursprünglicher Gegebenheit komme. Wäre das der Fall, wäre das Eigenwesentliche des Anderen in direkter Weise zugänglich, so wäre es bloß Moment meines Eigenwesens, und schließlich er selbst und ich selbst einerlei. Es verhielte sich ähnlich mit seinem Leib, wenn er nichts anderes wäre als der *Körper*, der rein in meinen wirklichen und möglichen Erfahrungen sich konstituierende Einheit ist, meiner primordialen Sphäre zugehörig als Gebilde ausschließlich meiner *Sinnlichkeit*. Eine gewisse Mittelbarkeit der Intentionalität muß hier vorliegen, und zwar von der jedenfalls beständig zugrundeliegenden Unterschicht der *primordialen Welt* auslaufend, die ein *Mit da* vorstellig macht, das doch nicht selbst da ist, nie ein Selbst-da werden kann. Es handelt sich also um eine Art des *Mitgegenwärtig-machens*, eine Art *Appräsentation*."[36]

Im Gegensatz zur rein immanenten Erfahrung der Reflektion ist die Appräsentation transzendent: Durch sie kommt das andere Bewusstsein zur Geltung, wird einer phänomenologischen Betrachtung zugänglich. Eine soziologische Analyse dieser Appräsentationsbeziehungen wurde durch Alfred Schütz[37] und später durch seine Schüler Peter L. Berger und Thomas Luckmann[38] durchgeführt.

Alfred Schütz setzt sich in seinem Werk mit den erkenntnistheoretischen Grundlagen der Soziologie und soziologischer Methoden auseinander und entwirft eine solche schließlich ausgehend von der Phänomenologie. Insbesondere verbindet er die Arbeiten Max Webers[39] mit Edmund Husserls transzendentalem Idealismus. Schütz knüpft hier an Webers Konzept der verstehenden Soziologie und seinen Begrifflichkeiten von Handeln und subjektiv gemeintem Sinn an[40]. Insbesondere befasst sich Schütz in seinem frühen Werk *Der sinnhafte Aufbau der sozialen Welt* auch mit dem Problem des Fremdverstehens, also dem Nachvollzug subjektiv gemeinten Sinns im Handeln anderer.

Dieses Sinnverstehen wird am verstehenden Deuten von Anzeichen expliziert. Anzeichen sind konstituierte Gegenstände, die über sich selbst hinaus auf etwas Angezeigtes verweisen. Hier von Interesse sind dabei Anzeichen, die auf anderes Bewusstsein verweisen, wie etwa Bewegungen eines Menschen. Eine besondere Form von Anzeichen sind Zeichen; dies sind solche Anzeichen, denen innerhalb eines Zeichensystems eine Bedeutung zugeschrieben wird[41]. Beispiele für solche Zeichen sind Gesten, die Farben einer Ampel, formelle Kleidung oder das Freizeichen eines Telefons. Insbesondere gehört aber jede Form von Sprache zu den Zeichen. In *Der sinnhafte Aufbau der sozialen Welt wird* das Verhältnis von

36 Husserl, Edmund: Husserliana 1, Cartesianische Meditationen und Pariser Vorträge, Hrsg. Und eingeleitet von Stephan Strasser. Nijhoff, 1950. S. 139.

37 Im Aufsatz „Symbol, Wirklichkeit und Gesellschaft" in: Schütz, Alfred: Theorie der Lebenswelt 2. Die kommunikative Ordnung der Lebenswelt. UVK, 2003. S. 119-197.

38 Insbesondere Berger, Peter L., Luckmann, Thomas: The Social Construction of Reality. Penguin Books, 1966. S.43ff.

39 Schütz bezieht sich hierbei insbesondere auf: Weber, Max: Wirtschaft und Gesellschaft. Mohr Siebeck, 1922.

40 Schütz, Alfred: Der sinnhafte Aufbau der sozialen Welt. Eine Einleitung in die verstehende Soziologie. UVK, 2004. S. 85ff.

41 Ebda. S. 251.

Anzeichen und Angezeigtem noch nicht als Appräsentationsbeziehung beschrieben, diese Begrifflichkeit wird jedoch später im Aufsatz *Symbol, Wirklichkeit und Gesellschaft*[42] verwendet.

Das Fremdverstehen wird bei Schütz an Objektivationen fremden Bewusstseins vollzogen[43]. Diese Objektivationen können in der Wahrnehmung als Gegenstände konstituierte Handlungen oder Artefakte sein. Zu den Artefakten gehören neben den Zeichen auch hergestellte physische Gegenstände, sie sind als Erzeugnisse auch Zeugnis, also Anzeichen, für das fremde Bewusstsein. Hier wird zwischen zwei Formen von Sinn unterschieden, die verstanden werden können. Der subjektive Sinn verweist dabei auf die Bewusstseinserlebnisse, die mit der Handlung oder dem Herstellen des Erzeugnisses verknüpft sind. Beim Verstehen subjektiven Sinns versucht man z.B. die Zwecke, die dieses eine Bewusstsein mit der Herstellung verbindet, nachzuvollziehen. Objektiver Sinn dagegen idealisiert über den Gegenstand:

> „Der objektive Sinn eines uns vorgegebenen Erzeugnisses wird hingegen keineswegs als Zeugnis für ein besonderes Erleben eines besonderen Du gedeutet, sondern als bereits konstituierte und gesetzte, von jeder Dauer und jedem Sinnzusammenhang in einer fremden Dauer losgelöste, deshalb mit »allgemeiner Bedeutung« begabte Gegenständlichkeit erfaßt. Gewiß weist schon die Behauptung, das zu Deutende sei ein Erzeugnis, auf ein Du zurück, welches es erzeugte, aber bei der Frage nach dem objektiven Sinn bekommen wir jenes alter ego cogitans und sein Erleben gar nicht in den Blick;“[44]

In Anlehnung an Alfred Schütz‘ Analyse des Fremdverstehens wird in dieser Arbeit ein modifizierter Begriff der Appräsentation zur Konstitution von Computerprogrammen als Erfahrungsgegenständen verwendet. Zunächst einmal sind diese als menschengemachte Gegenstände offensichtlich im Sinne Schütz‘ Erzeugnisse, also Anzeichen für fremdes Bewusstsein. Jedoch muss diese bloße Erkenntnis, dass andere Menschen in irgendeiner Form an der Herstellung eines Artefakts beteiligt waren, mir noch gar keine genaueren Rückschlüsse auf diese erlauben. Quellcode kann automatisch generiert sein, die Herstellung kann von einem Menschen alleine, von einer festen Gruppe (z.B. einem Unternehmen) oder von einem Kollektiv wechselnder Akteure erfolgen, wie z.B. in der Beschreibung des Linux-Projekts durch Eric S. Raymond:

> „I believed that the most important software (operating systems and really large tools like the Emacs programming editor) needed to be built like cathedrals, carefully crafted by individual wizards or small bands of mages working in splendid isolation, with no beta to be released before its time.
>
> Linus Torvalds’s style of development—release early and often, delegate everything you can, be open to the point of promiscuity — came as a surprise. No quiet, reverent cathedral-building here — rather, the Linux community seemed to resemble a great babbling bazaar of differing agendas and approaches (aptly symbolized by the Linux archive sites, which would take submissions from *anyone*) out of which a coherent and stable system could seemingly emerge only

42 Schütz, Alfred: Theorie der Lebenswelt 2. Die kommunikative Ordnung der Lebenswelt. UVK, 2003. S. 119-197.

43 Schütz, Alfred: Der sinnhafte Aufbau der sozialen Welt. Eine Einleitung in die verstehende Soziologie. UVK, 2004. S. 268.

44 Ebda. S. 271.

by a succession of miracles.“[45]

Wenn ein Programm, das ich verwende, gleichzeitig einen Basar unterschiedlicher und sich teilweise widersprechender Ansichten appräsentiert, so ist dies auch ein Deutungszusammenhang, unter dem ich das Programm betrachte. Umgekehrt appräsentiert der Quellcode für ein Anwendungsprogramm, das entwickelt oder überarbeitet wird, die Zusammenhänge, in denen diese Anwendung verwendet wird: Der Anwendungszusammenhang wird *mitgegenwärtig*, wenn der Sinn eines Programms als Anwendung *gesetzt* wird.

Hier sollen zwei Formen der Appräsentation unterschieden werden: Subjektive Appräsentation verweist auf die Deutungszusammenhänge, in denen andere sich auf den Gegenstand oder appräsentierte Gegenstände beziehen. In dieser Hinsicht appräsentiert ein Anwendungsprogramm subjektiv für Nutzer die Hersteller und umgekehrt appräsentiert für die Hersteller der Quellcode die Nutzer. Objektive Appräsentation dagegen bezeichnet mit dem Gegenstand verknüpfte leblose Gegenstände, die ich im Gegenstand zwar nicht direkt erfasse aber mir mitgegenwärtig machen kann. Ein ausführbares Anwendungsprogramm appräsentiert in dieser Hinsicht objektiv den Quellcode und umgekehrt.

Der Quellcode einer Programmbibliothek[46] appräsentiert wiederum objektiv anderen Quellcode, in dem auf die Funktionalitäten dieser Bibliothek zurückgegriffen wird und umgekehrt. In beiden Fällen bleibt der appräsentierte Gegenstand teilweise unbestimmt: Beim Schreiben der Bibliothek kann die Verwendung idealisiert erfasst sein; welche Anwendungen wie genau darauf zurückgreifen muss nicht vollständig bekannt sein. Umgekehrt abstrahiert die Verwendung der Bibliothek von der Implementierung der Funktionalität, deren Quellcode ist zwar mitgegenwärtig aber nicht notwendigerweise direkt zugänglich.

Diese Fassungen von objektiv und subjektiv sind nicht deckungsgleich mit dem objektiven und subjektiven Sinn bei Alfred Schütz: Der Bezug zu einem idealisierten Nutzer (z.B. „Wir gehen davon aus, dass diejenigen, die unsere Anwendung bedienen, die englische Sprache beherrschen.“) würde hier als subjektive Appräsentation aufgefasst. Eine idealisierte Verwendung gehört dagegen bei Schütz zum objektive Sinn eines Erzeugnisses.

Die objektive Appräsentation verweist bereits auf die Verknüpfung mehrerer Begriffsdimensionen unseres Programmbegriffs miteinander: Auch wenn Quellcode als Sprache gelesen wird ist dabei mitgegenwärtig, dass er physisch irgendwo gespeichert ist. Auch wenn ein Programm als Datei konstituiert wird (z.B. in der Handlung des Kopierens) kann dabei vergegenwärtigt werden, dass dieses Programm in einer bestimmten Programmiersprache eine formale Semantik hat.

Diese Appräsentationen werden hier nicht lediglich als Aspekte des subjektiven Zugangs zu Computerprogrammen begriffen. Die Formen der Appräsentation sind mithin gesellschaftlich bedingt. So ist das Verständnis, dass ein Datenträger wie das oben erwähnte SSD eben Information, mithin Dateien und im Besonderen auch Programme speichert, ein in gesellschaftlichen Prozessen reproduziertes und stabilisiertes. Ohne Techniken zum Auslesen der einzelnen Informationsbits, einer Deutung durch mehrere Ebenen von Zeichensystemen hindurch und schließlich einer Technik, diese Information menschlicher Erfahrung zugänglich zu machen, könnte das SSD nicht als Träger von Information verstanden werden. In einem

45 Raymond, Eric S,: The Cathedral and the Bazaar. Musings on Linux and Open Source by an Accidental Revolutionary. O'Reilly, 2001. S. 21f.

46 Programmbiliotheken sind Zusammenstellungen von Funktionen und Teilprogrammen, auf die in der Programmierung zurückgegriffen werden kann. Beispielsweise kann eine Algebra-Bibliothek eine Funktion zur Berechnung von Eigenwerten von Matrizen bereitstellen.

weiteren Schritt könnte den in einer Programmdatei vorgefundenen Buchstaben, Zahlen und Sonderzeichen auch keine formale Semantik zugeordnet werden ohne dass es eine gesellschaftlich stabilisierte Programmiersprache gibt, in der diesen Zeichen diese Semantik zukommt.

Eine soziologische Analyse dieser Verbindungen zwischen unterschiedlichen Bedeutungszusammenhängen oder heterogener Assoziationen nimmt Bruno Latour in seiner Grundlegung der Akteur-Netzwerk-Theorie vor. Latour versucht in dieser *relativistischen Soziologie* die Gruppenbildungs- und Auflösungsprozesse durch Selbst- und Fremdzuschreibung von Akteuren sowie die immer wieder performativ getätigten Bezüge zu Gruppierungen möglichst detailgetreu nachzuzeichnen. Neben den Gruppen bleiben auch die Handlungsträger zunächst unbestimmt: Handeln wird nicht als bewusster Akt von Subjekten, sondern als Konglomerat aus vielen verschiedenen Quellen betrachtet[47]. Hierfür wird der Begriff des Sozialen neu gefasst:

> „Er bezeichnet keinen Realitätsbereich und keinen bestimmten Gegenstand, sondern ist eher die Bezeichnung für eine Bewegung, eine Verschiebung, eine Transformation, eine Übersetzung, eine Anwerbung. Er bezeichnet eine Assoziation zwischen Entitäten, die in keiner Weise als soziale erkennbar sind, *außer* in dem kurzen Moment, in dem sie neu zusammengruppiert werden.“[48]

Eine Konsequenz aus der Perspektive, dass statt Gruppen Gruppierungsprozesse und Handlungsträger als Konglomerat betrachtet werden ist, dass leblose Gegenstände in den Handlungszusammenhängen, sofern sie diese mitbestimmen, als Akteure auftreten können[49]. Anders ausgedrückt gehen Handlungen nicht von Subjekten aus, sie kommen erst im Zusammenwirken heterogener Akteure zustande. Auf diese Arbeit bezogen kann beispielhaft ausgeführt werden: Das Programmieren eines (hinreichend komplexen) Programms ist nicht als Abfolge subjektiver Handlungen eines oder mehrerer Menschen alleine erklärbar. Die Tätigkeit wird erst durch die Verknüpfung dieser Menschen untereinander, mit Computern, Compilern, integrierten Entwicklungsumgebungen, dem Stromnetz, theoretischer Informatik, Datenträgern, Softwareentwicklungsprozessen, Programmiersprachen, Anwendungsfällen, User Stories, Bugtracking-Systemen, Debugging Tools, Unit Tests usw. in der eigentlichen Komplexität der Zusammenhänge erfasst, in der die Handlungen erfolgen.

Latour unterscheidet zwischen zwei Arten der Verbindung; zwischen Mittlern und Zwischengliedern[50]. Zwischenglieder transportieren Bedeutungen lediglich, ohne dass diese verändert werden. Mittler dagegen transportieren auch, jedoch als neue Quelle von Unbestimmtheit. Bedeutungen können verändert werden. Zur Illustration von Zwischengliedern kann die Übertragung einer E-Mail dienen. Diese läuft über viele verschiedene technische Zwischenglieder: über Modems, Kupferkabel, Glasfaserkabel usw. Sie wird an mehreren Stellen über mehrere Kommunikationsprotokolle codiert und decodiert, verfälscht und fehlerkorrigiert; am Ende dieser Kette kommt beim Empfänger die Nachricht ohne jegliche sichtbare Transformation an (die Bedeutung wird unverändert transportiert). Hier wird unter insgesamt erheblichem Aufwand sichergestellt, dass die Kette an sehr unterschiedlichen Techniken insgesamt als nicht-veränderndes Zwischenglied auftritt.

47 Latour, Bruno: Eine neue Soziologie für eine neue Gesellschaft. Einführung in die Akteur-Netzwerk-Theorie. Aus dem Englischen von Gustav Roßler. Suhrkamp, 2007. S. 77.

48 Ebda. S. 111f.

49 Ebda. S. 123.

50 Ebda. S. 69f.

Ein Mittler, der Bedeutung nicht immer und nicht immer unverändert transportiert ist in diesem Beispiel der natürlichsprachliche Inhalt der E-Mail. Die Bedeutung kann beim Lesenden unverändert, verfälscht oder gar nicht ankommen. Die Sprache fügt dem Übertragenen neben einem objektiven Sinn noch Stil, Metaphern oder Rechtschreibfehler hinzu. Die Nachricht kann missverstanden oder in einen völlig anderen Bedeutungskontext gesetzt werden. Am Beispiel der E-Mail kann gezeigt werden, wie aus Mittlern unter technischem Aufwand stabile Zwischenglieder werden: Die Fehleranfälligkeit der physischen Übertragung von Signalen wird in darüber liegenden Kommunikationsprotokollen durch Korrektur ausgeglichen, so dass trotz Rauschen der einzelnen Übertragungskanäle die E-Mail syntaktisch unverändert beim Empfänger ankommt.

Wie die Akteur-Netzwerk-Theorie für die hier durchgeführte Begriffsbestimmung fruchtbar wird kann wieder am Beispiel „Lesen eines Programms“ aufgezeigt werden. Wir betrachten hier die Übertragung ab der Anzeige eines einzelnen Schlüsselworts auf dem Monitor. Hier werden Lichtwellen erzeugt, die das menschliche Auge erfassen kann. Diese Lichtwellen werden in einem ersten Schritt als Buchstaben und zusammenhängend als Schrift gedeutet. Unter Rückgriff auf mehrere Kulturtechniken, die vereinfachend als „Lesen englischer Sprache“ zusammengefasst werden können, wird diese Schrift als „while“ gelesen – und zwar von links nach rechts und in gewisser Unabhängigkeit vom Schrifttyp: Sehr unterschiedliche Formen, Farben und Größen der Schrift werden als dasselbe Wort identifiziert. Die Sprachbeherrschung kann in diesem Beispiel als hoch voraussetzungsreiches Zwischenglied auftreten – ein Mittler vielfältiger möglicher Interpretationen der am Monitor angezeigten Formen, der durch normierte schulische Bildung als Zwischenglied stabil gehalten wird.

Noch interessanter für unser Vorhaben ist der zweite Schritt, in dem das Wort „while“ mit einer formalen Semantik in der verwendeten Programmiersprache verknüpft wird. Auch die Programmiersprache fungiert als Zwischenglied in dem Sinne, dass innerhalb einer Sprache der gleichen Syntax immer die gleiche formale Semantik zugeordnet werden muss. Auch dieses Zwischenglied wird über hohen Aufwand stabil gehalten: Formale Sprachspezifikationen werden geschrieben, veröffentlicht und verfügbar gehalten. Compiler, welche die Spezifikationen getreu umsetzen, werden geschrieben und stetig an neue Nutzungszusammenhänge angepasst, sodass sie mit neuer Hardware und neuer Software zusammen ihre Funktion erfüllen. Fehler bei der Umsetzung der Spezifikationen im Compiler werden sanktioniert und ausgebessert, durch Kritik, Bugreports, Forendiskussionen und neue Entwicklungsbestrebungen behoben.

Gleichzeitig bleiben die Sprachspezifikationen keineswegs immer gleich, auch wenn ein hoher Aufwand für Kompatibilität betrieben wird. Zur Veranschaulichung kann diese Beschreibung aus dem Vorwort einer neuen Version einer solchen Sprachspezifikation dienen, welche Änderungen an der Sprache mit aufgreift:

> „The language has grown a great deal in these past four years. Unfortunately, it is unrealistic to shrink a commercially successful programming language - only to grow it more and more. The challenge of managing this growth under the constraints of compatibility and the conflicting demands of a wide variety of uses and users is non-trivial. I can only hope that we have met this challenge successfully with this specification; time will tell.“[51]

Schon die Metaphorik, welche hier verwendet wird, zeigt Bewegung, sogar ein

51 Bracha, Gilad: Preface to the Third Edition. In: Gosling, James et al.: The Java Language Specification. Third Edition. Addison-Wesley, 2005. S. xxxi.

Eigenleben, an: Die Sprache wächst; es ist unmöglich, eine erfolgreiche Sprache zu schrumpfen. Die biologische Metapher des Wachstums illustriert eine relative Unbestimmtheit der Handlungsträger: Die Veränderung ist nicht das Ergebnis einer von einem einzelnen Menschen geplanten Handlung, sondern vielmehr aus dem Handeln vieler und den Voraussetzungen der vorherigen Sprachversion entstanden. Die Programmiersprache als Assoziation zweier unterschiedlicher Gegenstandsbereiche ist demnach keine zeitlos festgelegte endgültige Definition sondern sehr wohl abhängig von einem vielschichtigen Umfeld und von Veränderungen betroffen.

Mit diesen Überlegungen sind alle Werkzeuge vorhanden, die zur Begriffsentwicklung in dieser Arbeit herangezogen werden müssen. Ausgehend von den Phänomenen, auf die sich die Wahrnehmungsakte beziehen, wenn Computerprogramme als Gegenstand konstituiert werden, können die ersten vier Bedeutungsdimensionen des Begriffs herausgearbeitet werden: Das Programm als räumlich-zeitlicher, als syntaktischer, als semantischer und als eingebetteter Gegenstand. Die Verknüpfungen zwischen diesen Dimensionen werden als fünfte Dimension des Programmbegriffs untersucht: Das Programm wird als Assoziation betrachtet. Dabei ermöglicht die analytische Gliederung in Bedeutungsdimensionen eine hohe Präzision in der Begriffsanwendung. Die Betrachtung als Assoziation vermeidet eine Ausblendung der relevanten Zusammenhänge – die Assoziationen zwischen den ersten vier Bedeutungsdimensionen werden sich im weiteren Verlauf der Arbeit als essenzielle Bedingungen für Computertechnik überhaupt herausstellen. Bevor mit der eigentlichen Begriffsbestimmung (Kapitel 3 dieser Arbeit) begonnen wird soll das Vorhaben weiteren Begriffen im Umfeld von Computerprogrammen gegenübergestellt werden.

2.3 Verhältnis zu anderen Begriffen (Code, Software, Algorithmus, Artefakt)

Der in diesem Kapitel vorgestellte Zugang zu Computerprogrammen über die Erfahrung ermöglicht es, nach den Phänomenen separiert verschiedene Bedeutungsdimensionen des Begriffs zu unterscheiden, die verschiedene intentionale Zugänge zum Gegenstand erfassen. Gleichzeitig werden durch die Epoché, die Zurückhaltung in Urteilen, ontologische Fragen ausgeklammert. Zum Beispiel werden Fragen nach der Existenz einer formalen Semantik nach der Löschung aller auf sie verweisenden sprachlichen Gegenstände nicht verfolgt – sie haben für die Fassung des Erfahrungsgegenstands, der in irgendeiner Form sinnlich wahrgenommen werden kann, keine Auswirkungen. Das Verhältnis eines solchen Begriffs zu anderen Begriffen, auch anderen Fassungen des Programmbegriffs selbst, soll hier näher betrachtet werden.

Als *Software* können ähnliche Gegenstände bezeichnet werden wie diejenigen, die hier unter dem Programmbegriff erfasst werden. Neben Programmen können auch alle möglichen Formen von Daten als Software bezeichnet werden. Sprachlich unterscheidet sich Software von Programm unter anderem dadurch, dass letzteres pluralisierbar ist: man kann von mehreren Programmen sprechen, der Plural Softwares ist eher unüblich. Für bestimmte, abgrenzbare Gegenstände kann von einer Software oder einem Stück Software gesprochen werden. Während das Programm sprachlich eng verbunden mit der Tätigkeit des Programmierens ist, tritt Software insbesondere in der dichotomen Einteilung in Software und Hardware auf.

Hier liegt ein wesentlicher Unterschied in der Perspektive zum Vorhaben dieser Arbeit. Erstens ist die enge Verknüpfung mit allen Handlungen, die unter das Programmieren fallen, ein wichtiger Erfahrungsaspekt für die betrachteten Gegenstände. Die Betonung dieser Art des Zugangs hebt gleichzeitig die Gemachtheit von Programmen hervor, welche mit der hier verfolgten gesellschaftlichen Perspektive auf Programme verknüpft ist. Zweitens bringt die genannte Dichotomie von Software und Hardware erhebliche Probleme mit sich: Die Trennung ist durch Techniken wie programmierbare Hardware oder das Zusammenwirken von physischer und virtueller Räumlichkeit im Internet der Dinge nicht so scharf möglich, wie man zunächst annehmen könnte. Insbesondere aber ist auch jeder Zugang zu Software für die menschlichen Sinne stets physisch gegeben – im einfachsten Fall durch ein Blatt Papier, auf dem Programmiersprache lesbar ist. Während also eine Überschneidung der Begriffe Programm und Software vorliegt, zielt die vorliegende begriffliche Fassung auf Aspekte ab, die der Begriff Software weniger gut beleuchten kann.

Die Fassung von Programmierung als Handlung, welche grundsätzlich auf Programme als Gegenstand abzielt, hat Konsequenzen für den Programmbegriff selbst: Hardwarebeschreibungen, die in Formalsprache erfolgen, wären ebenfalls Programme. Der Begriff „programmierbare Hardware“, beispielsweise FPGAs[52], verwendet „Programmieren“ in dieser Hinsicht: Die in formalen Hardwarebeschreibungssprachen wie z.B. VHDL oder Verilog HDL formulierten sprachlichen Gegenstände sind keine Abfolge von Befehlen, die ein Prozessor der Reihe nach abarbeitet, sondern eine Beschreibung parallel ablaufender (zumeist, aber nicht immer, digitaler) Prozesse in einem integrierten Schaltkreis. Es handelt sich hierbei zwar um Grenzfälle des Programmierens, jedoch sind deutliche Parallelen im Entwurf von (digitalen) Schaltungen und dem Schreiben eines Programms in einer üblichen Programmiersprache zu erkennen.

52 Field Programmable Gate Arrays, ein Typ von Chip, dessen Schaltkreisverbindungen rekonfigurierbar sind – auf diese Weise kann der Chip selbst „programmiert“ werden und nicht lediglich Programme ausführen.

Als Code, Source Code, Quellcode oder Quelltext wird ein sprachlicher Gegenstand bezeichnet, der in einer bestimmten Programmiersprache verstanden werden kann. In einer vereinfacht dargestellten, idealtypischen Softwareentwicklung wird der Quellcode durch Menschen geschrieben, anschließend durch einen Compiler in Maschinencode übersetzt und kann danach direkt ausgeführt werden[53]. Maschinencode ist im Gegensatz zum Quellcode üblicherweise maschinenspezifisch, das heißt vom Befehlssatz der Hardware abhängig und nur unter sehr großem Aufwand menschenlesbar. Quellcode ist dagegen in einer höheren Programmiersprache formuliert, die Menschen zugänglicher ist. Er ist üblicherweise nicht ohne weitere Programme (Compiler, Interpreter) ausführbar. Der Quellcode wird in dieser Arbeit als ein zentraler sprachlicher Zugang zu Computerprogrammen erfasst, die zugehörige Begriffsdimension ist das Programm als syntaktischer Gegenstand. Einige bestehende Begriffe von Computerprogrammen setzen Code und Programm gleich, beispielsweise die bereits zitierte ISO/ANSI-Definition von Programmen:

> „[a] syntactic unit that conforms to the rules of a particular programming language and that is composed of declarations and statements or instructions needed to solve a certain function, task, or problem“[54]

Diese Definition zielt also im hier entwickelten Begriff auf eine unter mehreren Bedeutungsdimensionen ab. Ein sprachlicher Zugang zu Programmen als syntaktischen Gegenständen findet genau über die genannten Deklarationen, Statements und Instruktionen statt.

Unter Algorithmus wird eine formale oder formalisierbare Rechenvorschrift verstanden, mit der eine bestimmte Aufgabe gelöst werden kann. Der Begriff ist eng verknüpft mit der Eigenschaft der Berechenbarkeit. Ein Algorithmus zerlegt eine klar definierte berechenbare Aufgabe in wiederum klar definierte berechenbare Teilschritte, sodass die gesamte Berechnung formal erfasst wird. Auf ein Programm bezogen wird mit Algorithmus dabei nicht der Code selbst bezeichnet, sondern seine formale Semantik als mathematisches Objekt. Es existieren verschiedene Notationsweisen für Algorithmen. Diese sind zueinander äquivalent, sie können die gleiche Klasse von Funktionen ausdrücken, und zwar die Klasse aller (intuitiv) berechenbaren Funktionen. (Intuitive) Berechenbarkeit ist nach der Church-Turing-These gleichbedeutend damit, dass eine Turingmaschine, welche diese Funktion berechnet, definiert werden kann. Die Äquivalenz verschiedener formaler Darstellungsweisen wie der Turingmaschine, dem λ-Kalkül und μ-rekursiven Funktionen wurde von Alonzo Church, Stephen Cole Kleene und Alan Turing nachgewiesen[55]. Zwei weitere äquivalente Darstellungsweisen sind die von Emil Leon Post beschriebenen Post-Turing-Maschinen und Registermaschinen.

Ein Programm in seiner Ausführung auf einem Computer wird auch als Prozess (oder Task) bezeichnet. Im engen Sinne bezeichnet Prozess damit die Kopie vom Maschinencode, die als Reihe von Instruktionen vom Prozessor ausgeführt wird, sowie direkt damit verknüpfte Daten, wie beispielsweise momentane Inhalte von Variablen. Abhängig von der verwendeten Sprache können diese Daten beispielsweise in den zwei unterschiedenen Speicherbereichen

53 Diese vereinfachte Darstellung abstrahiert stark. In tatsächlicher Softwareentwicklung kommen regelmäßig zahlreiche weitere Elemente zur Geltung, etwa Entwicklungsumgebungen, Laufzeitumgebungen, Testsysteme, Bibliotheken, Buildsysteme, Codegeneratoren, virtuelle Maschinen, statisches und dynamisches Linken usw.

54 International Organization for Standardization. (2015): Information technology - Vocabulary. (ISO/IEC Standard 2382:2015(en)). Retrieved from https://www.iso.org/obp/ui/#iso:std:iso-iec:2382:ed-1:v1:en (abgerufen am 29.06.2018).

55 Zu einer historischen Übersicht der Diskussionen, die zur Church-Turing-These geführt haben siehe Soare, Robert I.: Turing Oracle Machines, Online Computing, and Three Displacements in Computability Theory. In: Annals of Pure and Applied Logic Vol. 160 Issue 3. 2009. 368-399.

Stack und Heap festgehalten werden. Ein einzelner Prozess kann in mehrere voneinander teilweise entkoppelte Threads aufgeteilt sein. Die Verwaltung von Prozessen und Threads wird durch das Betriebssystem vorgenommen[56]. Für den hier gewählten phänomenologischen Zugang sind Prozesse ein wesentlicher Teil der Wahrnehmung von ausgeführten Computerprogrammen. Da Handlungen mit intentionalem Bezug zu ausgeführten Programmen jedoch deutlich mehr Anknüpfungspunkte umfassen als die Relation zum Betriebssystem, wird Prozess hier in seiner alltagssprachlichen, wesentlich umfangreicheren, Bedeutung verwendet. Prozess im engen Sinne, der vom Betriebssystem verwaltete Gegenstand, ist lediglich ein Teil dieser Bedeutungsdimension.

Der hier gewählte phänomenologische und soziologische Zugang zu Computerprogrammen erfasst diese als technische Artefakte: Sie sind in einer zumindest mittelbar menschlich gemachten physischen Form erfahrbar und sie haben eine Funktion. Raymond Turner erörtert in einem Aufsatz, inwiefern das gleiche für Programmiersprachen gilt. Technische Artefakte sind dabei insbesondere durch die Verknüpfung von Struktur und Funktion definiert:

> „Technical artifacts include all the common objects of everyday life such as televisions, computer keyboards, paper clips, telephones, smart phones, dog collars, and cars. They are intentionally produced things. This is an essential part of being a technical artifact. For example, a physical object that accidently carries out arithmetic is not by itself a calculator. This teleological aspect distinguishes them from other physical objects.
>
> Natural objects just happen. Descriptions of them as kinds, or in the case of biological objects in terms of their function, are post hoc and best seen as theories about them. Moreover, technical artifacts can only fulfill their intended function because of their actual physical structure. This is quite different to social constructions such as money where there is no necessary connection between function and physical structure.“[57]

Programmiersprachen sind hierbei ausschließlich deshalb technische Artefakte, weil sie ohne eine physische Implementierung durch Interpreter, Compiler, virtuelle Maschinen, etc. wirkungslos sind[58] und gerade nicht als Sprache zum Programmieren verwendet werden können. Ausschlaggebend ist dabei immer die aufeinander bezogene Dualität von physischer Struktur und Funktion. Diese beiden Aspekte des technischen Artefakts sind durch Intention verbunden: Das mathematische Objekt kann auch unabhängig von einer physischen Implementierung untersucht werden, aber sobald es durch menschliche Intention als normative Vorgabe für Compiler etc. verwendet wird, kann die Sprache als in diesem Sinne dualistisches technisches Artefakt gelten[59].

Zu Turners begrifflicher Fassung technischer Artefakte unterscheidet sich der Begriffszugang zu Programmen in dieser Arbeit in zwei wesentlichen Punkten. Erstens trennt dieser Ansatz durch die phänomenologische Perspektive noch zwei weitere

56 Threads werden hier im engen Sinne, als Kernel Level Threads aufgefasst. User Level Threads werden nicht vom Betriebssystem verwaltet.

57 Turner, Raymond: Programming Languages as Technical Artifacts. In: Philosophy & Technology Vol. 27 Iss. 3. 2014. S. 379.

58 Denkbar ist auch, dass ein Mensch das Programm im Kopf oder auf Papier ausführt, *Maschine spielt*. Dies dient z.B. der Fehlersuche. In diesem Fall kommt die Programmiersprache nicht als *technisches* Artefakt zur Anwendung.

59 Siehe Turner, Raymond: Programming Languages as Technical Artifacts. In: Philosophy & Technology Vol. 27 Iss. 3. 2014. S. 393f.

Bedeutungsdimensionen von Programm ab: Turners dualistische Fassung technischer Artefakte unterscheidet zwischen Physik und Funktion. Was hier als sprachlich-syntaktisches Phänomen begriffen wird, fällt bei Turner auch unter die physische Implementierung. Und die Funktion eines Programms, sein Laufzeitverhalten, wird sowohl von der Implementierung (als Code) als auch von der formalen Semantik unterschieden. Zweitens erfasst der gesellschaftliche Blickwinkel dieser Arbeit die Zusammenhänge zwischen den unterschiedlichen Bedeutungsdimensionen, die Assoziationen, mit einem anderen Schwerpunkt als Turners ontologische Überlegungen: Programmiersprachen und ihre Implementierungen sind unter hohem Aufwand stabilisierte Zwischenglieder zwischen Physik, Syntax und Semantik der Programme. Durch die zusätzliche Ebene der Syntax kann hier genauer unterschieden werden: Eine Programmiersprache ist Verbindung von Programmen als syntaktischen und Programmen als formal-semantischen Gegenständen. Ihre Implementierung, z.B. als Interpreter oder als Compiler mit zugehöriger (virtueller oder physischer) Maschine, ist die Verbindung zwischen dem syntaktischen Programm und dem ausführbaren Programm. Als technische Artefakte werden hier sowohl Programme in jeder Form als auch die Programmiersprachen und Compiler begriffen: Auch eine formale Semantik oder eine Spezifikation kann nur über den Umweg der Physik wahrgenommen werden und Compiler erscheinen uns selbst als Programme mit einer bestimmten Funktion.

Eine wesentliche Eigenschaft technischer Artefakte ist, dass sie im Hinblick auf ihre Funktion konzipiert werden. Diese Konzeption ist nicht eindeutig durch die Funktion vorgegeben, es gibt verschiedene Möglichkeiten, wie Artefakte im Allgemeinen und Computerprogramme im Speziellen ihre Aufgabe erfüllen können. Zusammen mit unterschiedlichen auf Artefakte bezogenen Bedeutungszuschreibungen nennen Wiebe E. Bijker und Trevor J. Pinch diese grundsätzliche Offenheit der Konzeption die interpretative Flexibilität:

> „[...]the demonstration that technological artifacts are culturally constructed and interpreted; in other words, the interpretative flexibility of a technological artifact must be shown. By this we mean not only that there is flexibility in how people think of or interpret artifacts but also that there is flexibility in how artifacts are *designed*. There is not just one possible way or one best way of designing an artifact.“[60]

Diese interpretative Flexibilität existiert bei Computerprogrammen in mehrfacher Hinsicht, insbesondere auf das verwirklichte Design bezogen. In den Fällen, in denen völlig bewusst aus mehreren Möglichkeiten der Realisierung ausgewählt wird, präsentiert sich diese Flexibilität in der Form von Designentscheidungen. Häufig finden solche Entscheidungen bereits statt, bevor die erste Zeile Code geschrieben wurde: Welche Programmiersprache wird verwendet, wird ein detailliert gestalteter Softwareentwicklungsprozess verwendet, wird der Quellcode offengelegt, wer arbeitet alles am Programm, wie wird mit Fehlern umgegangen, ...? Für jede dieser Entscheidungen können Entscheidungsfindungsprozesse und oft auch Legitimationsprozesse untersucht werden.

Die Designentscheidungen beziehen sich auf sehr unterschiedliche Aspekte der jeweiligen Programme. Die Auswahl einer verwendeten Programmiersprache setzt Überlegungen voraus, die sich deutlich von denen bei der Anordnung verschiedener Elemente einer grafischen Benutzeroberfläche unterscheiden. Eine Stärke des hier vorgeschlagenen Programmbegriffs ist dabei, dass auf Programme bezogene Designentscheidungen nach den

60 Pinch, Trevor J., Bijker, Wiebe: The Social Construction of Facts and Artifacts: Or How the Sociology of Science and the Sociology of Technology Might Benefit Each Other. In: Bijker, Wiebe E. et al.: The social constructions of technological systems: New directions in the Sociology and History of Technology. MIT Press, 1993 (1987). S. 40.

intendierten Gegenständen unterschieden werden können: Die Entscheidungen beziehen sich jeweils auf das Programm als physisch gemeinten Gegenstand (Wo wird es gespeichert? Wie wird es verteilt? Für welche Art von Gerät? Mit welcher Peripherie interagiert es?), als syntaktisch-sprachlichen Gegenstand (Welche Versionierung, welches Versionsmanagement wird eingesetzt? Wie wird der Code geordnet und lesbar gehalten?), als semantischen Gegenstand (Welcher Algorithmus wird verwendet? Wie werden die Anforderungen formalisiert?) usw. Der phänomenologische Zugang zu Programmen als Artefakten ist hier ein fruchtbarer Ausgangspunkt für weitere Überlegungen zu Gestaltungs- und Aushandlungsprozessen bezüglich der Technik.

Eine begriffliche Fassung von Programmen als abstrakten Artefakten stellt Nurbay Irmak[61] vor. Der Ansatz stellt sich auf den ersten Blick ähnlich zum Vorhaben dieser Arbeit dar, Irmak unterscheidet zwischen den vier Begriffen *Algorithm, Code, Copy* und *Execution*. Diese können, wenn auch nicht deckungsgleich, den Bedeutungsdimensionen des Programms als semantischem, syntaktischen, räumlich-zeitlichen und eingebetteten Gegenstand zugeordnet werden. Irmak nimmt diese vier Klassen von Gegenständen auf und legt dar, dass Programme nicht durch einen einzelnen dieser Gegenstände charakterisiert werden können, um anschließend eine ontologische Bestimmung von Programmen als abstrakten Artefakten vorzunehmen, die zwar zeitlich – aber nicht räumlich – gegeben sind. Irmak illustriert diese Bestimmung mit Parallelen zwischen Programmen und musikalischen Werken.

In dieser Arbeit wird die ontologische Frage ausgeklammert. Stattdessen werden Phänomene betrachtet und jeweils charakterisiert, ob diese als räumlich und/oder zeitlich intendiert auftreten. Die Bedeutungen überschneiden sich zwar auf den ersten Blick, sind aber bei näherer Betrachtung durchaus unterschiedlich. Insbesondere fokussiert diese Arbeit die gesellschaftlich produzierte Assoziation zwischen den verschiedenen Bedeutungsdimensionen, während sie für die ontologischen Überlegungen zu Programmen eine eher untergeordnete Rolle spielt.

Timothy Colburn und Gary Shute befassen sich in ihrem Aufsatz *Abstraction in Computer Science*[62] mit dem Verhältnis zwischen Informatik und Mathematik. Dabei kommen sie zu dem Ergebnis, dass die beiden Wissenschaften durch die Formen von Abstraktion, die verwendet werden, unterscheidbar sind. Mathematische Abstraktion wird durch *information neglect* charakterisiert, Informatik durch *information hiding*. Programme treten dabei als wesentliche Abstraktion von physischen Prozessen auf:

> „Whatever the elements of computational processes that are described in textual programs, however, they are never the actual, micron-level electronic events of the executing program; textual programs are always, no matter what their level, *abstractions* of the electronic events that will ultimately occur. Since it is a practical impossibility for a textual program to describe the electronic events, every such program describes a computational process as an abstraction that hides information or details at a lower level.“[63]

Die Abgrenzung der Informatik zur Mathematik ist dabei nicht scharf zu verstehen, da insbesondere in der theoretischen Informatik auch *information neglect*-Abstraktion betrieben wird. Colburns und Shutes Argument ist jedoch, dass *information hiding*, also die Abstraktion von Dingen, die zwar nicht vernachlässigt, aber z.B. automatisiert bearbeitet oder in einer

61 Irmak, Nurbay: Software is an abstract artifact. Grazer Philosophische Studien Vol. 86 Issue 1. 2012. S. 55-72.

62 Colburn, Timothy, Shute, Gary: Abstraction in computer science. Minds and Machines Vol. 17 Issue 2. 2007. S, 169-184.

63 Ebda. S. 177.

anderen Protokollschicht behandelt werden, ein zentraler Aspekt in der Entwicklung von Computerprogrammen ist. Mathematische Abstraktion ist im Gegensatz dazu dadurch charakterisiert, dass die ausgesparte Information für die weitere Arbeit nicht mehr relevant ist. Nach Colburn und Shute ist Softwareentwicklung ohne *information hiding* aufgrund der Komplexität moderner Computersysteme grundsätzlich unmöglich:

> „The complexity of behaviour of modern computing devices makes the task of programming them impossible without abstraction tools that hide, but do not neglect, details that are essential in a lower-level processing context but inessential in a software design and programming context.“[64]

Aus Sicht dieser Arbeit sind an dem Aufsatz zwei Dinge von besonderer Relevanz: Erstens spielt die beschriebene Abstraktionsform für Computerprogramme sowohl innerhalb einzelner Bedeutungsebenen als auch in den Verknüpfungen dazwischen eine gewichtige Rolle: Die Möglichkeit höherer Programmiersprachen, von konkreten physischen Maschinen zu abstrahieren, ist ein wesentlicher Grund für deren Verwendung. Zweitens berührt Colburns und Shutes Arbeit die Fragestellung nach dem Verhältnis von Programmen zur Wissenschaft, insbesondere zur Informatik und zur Mathematik. Charakteristisch für die Informatik ist dabei, dass sehr unterschiedliche und verschieden stark formalisierte wissenschaftliche Methoden angewendet werden.

Die Teilgebiete der Informatik zielen mitunter auf verschiedene Gegenstände ab. So beschäftigt sich die theoretische Informatik, z. B. in der formalen Verifikation, Komplexitäts- und Berechenbarkeitstheorie, hauptsächlich mit der formalen Semantik von Programmen. Softwaretechnik zielt mitunter auf den Code als syntaktischen Gegenstand ab, aber auch auf soziale Prozesse, die zur Herstellung von Code führen; Entwicklungsprozesse und Design Patterns geben Hilfestellung bei der Überführung einer Semantik in Code. Der Entwurf digitaler Schaltungen und Rechnerarchitektur werden unter anderem zur Herstellung physischer Gegenstände wie beispielsweise Prozessoren verwendet.

Die beiden vorgestellten Aufsätze von Turner, Colburn und Shute sehen über die Teildisziplinen hinweg die zentrale Aufgabe der Informatik in der Produktion von Software[65] bzw. von technischen Artefakten[66]. Durch diese Zielsetzung lassen sich viele der Bestrebungen in der Informatik als beispielhaft für die Technowissenschaften charakterisieren, Wissenschaft und Technik sind eng miteinander verflochten. Daher kommt im Vergleich zur idealtypischen Vorstellung von Naturwissenschaften auch anderen Formen der Wissensproduktion und Wissenskommunikation in der Informatik eine zentrale Bedeutung zu; viele der Erkenntnisse der Informatik können als technowissenschaftliches Wissen verstanden werden. Alfred Nordmann analysiert diese Wissensform anhand wissenschaftlicher Veröffentlichungen:

> „When one speaks of scientific knowledge, one usually means what is written down in textbooks or what represents the current consensus of scientists on a certain topic. In both cases, the knowledge that is produced by scientists consists in statements that are determined to be true or empirically adequate or rather likely to be true. Normally these statements get written down as theories or hypotheses, explanations and descriptions. In contrast, technoscientific knowledge consists in the acquisition and demonstration of basic capabilities of visualization, manipulation, modelling, or construction. The typical scientific publication argues

64 Colburn, Timothy, Shute, Gary: Abstraction in computer science. Minds and Machines Vol. 17 Issue 2. 2007. S. 176.

65 Ebda S. 173.

66 Turner, Raymond: Programming Languages as Technical Artifacts. In: Philosophy & Technology Vol. 27 Iss. 3. 2014. S. 395.

> that ‘here is evidence to confirm or disconfirm an hypothesis.’ The typical technoscientific publication shows that ‘here is what we accomplished in our laboratory.’ For example, it is a major achievement in nanotechnoscience to do something at room temperature and in atmospheric conditions where others required extreme coldness in a vacuum.“[67]

In der Informatik kommen sehr unterschiedliche Wissensformen zusammen. Zunächst gibt es formalwissenschaftliches Wissen insbesondere in der theoretischen Informatik, dieses ist analog zur Mathematik beweisbar und bedarf keiner empirischen Aussagen oder Demonstrationen; ein Beispiel wären komplexitätstheoretische Aussagen. Empirisches Wissen in Form falsifizierbarer Aussagen spielt auch eine Rolle, beispielsweise die Ergebnisse von mit sozialwissenschaftlichen Methoden konzipierten Usability-Studien, Kenntnisse über die Größe von Datenströmen oder die Verbreitung von Software mit einer bestimmten Sicherheitslücke. Insbesondere ist jedoch die von Nordmann als technowissenschaftlich bezeichnete Form von Wissen relevant: Das Vermögen, unter den verfügbaren (insbesondere technischen) Voraussetzungen bestimmte Phänomene zu erzeugen, zu stabilisieren und zu kontrollieren. Die Veröffentlichung eines *proof of technology/concept*, prototypische Implementierungen, beispielhafte Anwendungen und Demonstrationen der Fähigkeiten, die mit neuen Techniken in einem gegebenen Umfeld erreicht werden können und die Kommunikation über diese Machbarkeiten und Fähigkeiten sind für die Wissensproduktion in der Informatik unverzichtbar.

Die Informatik kann auch beispielhaft für andere Technowissenschaften in Hinsicht auf die Bedingungen und die Bedeutung dieser Wissensform untersucht werden. Zunächst demonstriert die Softwareentwicklung als zentrale Methode der Wissensproduktion die Voraussetzung handwerklichen *Dingwissens*: Ein Forschungsprojekt, in dem beispielsweise neue Optimierungen für Compiler entwickelt und beispielhaft angewandt werden, ist hoch voraussetzungsreich im Bezug auf Teamarbeit, Kenntnisse in Programmiertechniken, Sprachen, Designprozessen, Compilerbau, Softwarewartung usw. Am Beispiel maschinellen Lernens zeigt sich die Relevanz des technischen Umfelds: Die Forschungserfolge auf dem Gebiet sind eng verknüpft mit der Entwicklung der Fertigungsverfahren für Prozessoren. Die Demonstration der Fähigkeiten von *Deep Learning*-Algorithmen und auch die Anwendbarkeit des Wissens ist direkt von der Verfügbarkeit von Prozessoren abhängig, die die notwendigen Berechnungen in akzeptabler Zeit durchführen können.

Ein weiteres Beispiel für die Abhängigkeit der Forschung in der Informatik vom technischen Umfeld ist die Forschung in der Cybersicherheit: Demonstrationen von Schwachstellen in Software oder Hardware beziehen sich gerade auf die Techniken, die momentan angewendet werden. Ein sicheres Netzwerkprotokoll wird fast immer in Bezug und in Abhängigkeit zu bestehenden und angewendeten Technologien entwickelt. Eine offene Frage in diesem Bereich ist die, ob die Informatik eine eigene auf ihr Gebiet bezogene Abschätzung der Technikentwicklung betreibt: Welche Rolle spielen erwartete Eigenschaften und insbesondere Auswirkungen von Technologien, die noch nicht existieren? Wenn Quantenalgorithmen entwickelt werden, bevor es anwendbare Quantencomputer gibt, oder (spezifischer auf die Auswirkungen bezogen) vorsorglich eine Post-Quanten-Kryptographie entsteht, wenn Privatsphäre und persönliche Daten in der personalisierten Medizin abgesichert werden – ohne dass es bereits angewendete personalisierte Medizin gibt – wie lässt sich dieses generierte Wissen charakterisieren?

Im Bezug auf den Programmbegriff lässt sich festhalten, dass bei den intentionalen

67 Nordmann, Alfred: Philosophy of technoscience in the regime of vigilance. In: Hodge, Graeme A., Bowman, Diana M., Maynard, Andrew D.: International handbook on regulating nanotechnologies. Edward Elgar Publishing, 2010. S. 32f.

Bezügen zu diesen Gegenständen insbesondere zwei Wissensformen charakteristisch sind: *technowissenschaftliches Wissen* und *Dingwissen*[68]. Dingwissen zeichnet sich im Vergleich der beiden Wissensformen dadurch aus, dass die Wissensdemonstration im Artefakt selbst liegt:

> „[...]claims that are variants of "thing knowledge" and that belong in the realm of technology and engineering rather than science or technoscience. In contrast to knowledge of control that is established by way of technoscientific research, these include the knowledge of handling and use, the knowledge of making, building, repairing, and the knowledge of design. And whereas the claim of having achieved a capability of control consists in a publicly certified report that this achievement can be and has been demonstrated, the variants of thing knowledge are established by way of artifactual demonstrations. Here it is not only some report but the structures, devices, artifacts themselves that embody intersubjectively accessible evidence of achieved making, building, or design: engineers managed to build this dome, or the circuitry on a chip relates inputs to outputs causally and formally."[69]

Durch die Möglichkeit, Quellcode und auch anwendbare übersetzte Programme extrem schnell und unkompliziert auszutauschen, ist die Wissensdemonstration durch Artefakte in der Informatik tatsächlich leichter durchführbar als in anderen Technowissenschaften. Dennoch sind die als technowissenschaftlich charakterisierten Veröffentlichungen und Berichte relevant: Aussagen über die verringerte Laufzeit bei der Anwendung einer neuen Optimierung, bessere Lernergebnisse bei Einsatz eines neuen Lernalgorithmus, die Leistung neuer Komprimierungsverfahren usw. werden im wissenschaftlichen Betrieb über Veröffentlichungen kommuniziert, der Austausch von Quellcode ist eher als zusätzliche Wissensdemonstration aufzufassen. Die Veröffentlichung kann zwar sehr wohl auch einen (abstrakten) Quellcode enthalten, jedoch ist dieser üblicherweise in eine Beschreibung dessen, was durch den Code erreicht wird, eingerahmt.

Durch den Bezug phänomenologischer Methoden auf Computerprogramme weist diese Arbeit Parallelen zu postphänomenologischen Untersuchungen auf. In den maßgeblich durch Don Ihdes *Technology and the Lifeworld*[70] und *Postphenomenology: Essays in the Postmodern Context*[71] geprägten Analysen werden phänomenologische Methoden auf Wissenschaft und Technik angewandt, um deren Bedeutung für Menschen und menschliche Bezüge zur Welt herauszuarbeiten. Auch hier sind die Gemachtheit technischer Gegenstände sowie die Verhältnisse zwischen Mensch, Technik und Gesellschaft zentrale Aspekte der philosophischen Überlegungen. Robert Rosenberger und Peter-Paul Verbeek beschreiben das postphänomenologische Programm anhand von Beispielen:

> „As "empirical philosophy," postphenomenology does not base itself on the philosophical tradition and on conceptual analysis only, but also on the study of actual technological practices and artifacts. In doing so, it does not merely "apply" philosophical analyses to science and technology, but it investigates the implications of such practices and artifacts for philosophical conceptualizations. Rather than analyzing the imaging devices on the Mars orbiter in terms of existing

68 Das heißt nicht, dass die anderen Wissensformen irrelevant sind. Wie bereits erläutert kommen auch empirisches Wissen und formalwissenschaftliches Wissen in der Herstellung von Programmen zum Einsatz.

69 Nordmann, Alfred: Object lessons: towards an epistemology of technoscience. In: Scientiae Studia Vol. 10. 2012. S. 28f.

70 Ihde, Don: Technology and the Lifeworld: From Garden to Earth. Indiana University Press, 1990.

71 Ihde, Don: Postphenomenology: Essays in the Postmodern Context. Northwestern University Press, 1993.

> theories of perception or knowledge development, it investigates how this device establishes a new kind of sensorial relationship between scientists and the planet Mars, which is a new basis for knowledge development. And rather than assessing obstetric ultrasound in terms of pre-given normative frameworks it investigates how this technology urges us to conceptualize the way technologies help to shape normative frameworks, and to make room for a moral significance of technology in ethical theory."[72]

Der hier verfolgte Ansatz zur Begriffsbestimmung würde also nach der zitierten Beschreibung selbst noch nicht unter Postphänomenologie fallen, auch wenn der entwickelte Begriff für nachfolgende postphänomenologische Untersuchungen anwendbar ist. Diese Arbeit hat nicht zum Ziel, die Auswirkungen von Computerprogrammen auf bestehende philosophische Diskurse z.B. über die Subjektkonstitution[73] zu untersuchen. Ein solches Unterfangen wäre eine nachfolgende Untersuchung, die auf den hier entwickelten Programmbegriff zurückgreifen kann.

Abschließend soll hier noch kurz auf das Verhältnis des entwickelten Programmbegriffs zu Gegenständen eingegangen werden, welche sprachlich als Programm bezeichnet werden, mit Computern unmittelbar jedoch nichts zu tun haben: Forschungsprogramme, Abendprogramme, Parteiprogramme, Förderprogramme etc. sind hier Beispiele. Das Vorhaben dieser Arbeit zielt nicht darauf ab, einen allgemeinen Programmbegriff zu entwickeln; sowohl das Vorgehen als auch die entwickelten Bedeutungsdimensionen des Begriffs sind spezifisch auf Computerprogramme zugeschnitten. Trotzdem stellt sich heraus, dass die Begriffsstruktur mit Anpassungen auch auf andere Programme übertragbar sein kann. Aus diesem Grund soll im Ausblick (Kapitel 5.2 dieser Arbeit) der entwickelte Begriff anderen Programmen gegenübergestellt werden. So kann dort erörtert werden, inwiefern der Begriff für Computerprogramme auch einen Beitrag zu einem allgemeinen Programmbegriff leisten kann.

72 Rosenberger, Robert, Verbeek, Peter-Paul: A field guide to postphenomenology. In: Rosenberger, Robert, Verbeek, Peter-Paul: Postphenomenological investigations: Essays on human-technology relations. Lexington, 2015. S. 30f.

73 Vgl. Kapitel 5 in Ihde, Don: Technology and the Lifeworld: From Garden to Earth. Indiana University Press, 1990. S. 72-123.

3. Ein mehrdimensionaler Programmbegriff

Die im letzten Kapitel identifizierten Bedeutungsdimensionen werden an dieser Stelle ausgearbeitet und erweitert zu einem mehrdimensionalen Programmbegriff. Die ersten vier Bedeutungsdimensionen stehen dabei für vier unterschiedliche Formen, in denen das Computerprogramm als Gegenstand konstituiert wird. Nachdem die Dimensionen einzeln beschrieben werden, wird der innere Zusammenhang des Programmbegriffs – die Verhältnisse zwischen den unterschiedlich konstituierten Gegenständen – herausgearbeitet und zu einer fünften Bedeutungsdimension entwickelt. Die Struktur dieser Arbeit orientiert sich an der Struktur des entwickelten Begriffs: Kapitel 3.1 bis 3.4 behandeln jeweils eine der ersten vier Bedeutungsdimensionen. Unterkapitel kennzeichnen eine mögliche genauere Unterscheidungen der Gegenstandskonstitutionen: In Kapitel 3.1 wird z.B. das Programm als räumlich-zeitlicher Gegenstand erfasst. Kapitel 3.1.1 und 3.1.2 beschreiben zwei unterscheidbare Formen räumlich-zeitlich konstituierter Programme. In Kapitel 3.5 werden die Zusammenhänge zwischen den ersten vier Bedeutungsdimensionen des Begriffs als weitere fünfte Bedeutungsdimension entwickelt. Kapitel 3.6 schließt diesen zentralen Teil der Arbeit mit einem Exkurs darüber ab, inwiefern Computerprogramme als Kompositionen verstanden werden können.

Zunächst kann ein Computerprogramm als räumlich-zeitlicher Gegenstand (siehe Kapitel 3.1) konstituiert werden. Diese Art von Gegenstand ist dadurch charakterisiert, dass Programme intentional als zu einer bestimmten Zeit als an einem bestimmten Ort befindlich aufgefasst werden. Als physisch intendierte Gegenstände (siehe Kapitel 3.1.1) bilden diese die Grundlage für die sinnliche Wahrnehmung von Programmen: Alles, was direkt gesehen, gehört oder ertastet werden kann, wird als zu einer Zeit an einem Ort im Raum wahrgenommen. Das trifft auf Ausdrucke, Datenträger, Laptops usw. zu, aber auch beispielsweise auf die Ausgabe auf einem Monitor. Eine weitere Menge von Gegenständen, die als räumlich-zeitlich intendiert werden, sind virtuell-physisch (siehe Kapitel 3.1.2). Diese Gegenstände erscheinen auch zu einer bestimmten Zeit an einem bestimmten Ort, allerdings fallen die Vorstellungen von Ort und Zeit nicht mit den physischen Vorstellungen zusammen. Eine Programmdatei, die innerhalb eines Dateisystems verortet oder über eine URL[74] lokalisiert wird erscheint uns an einem bestimmten Ort – in einem Ordner, auf einer Website – aber dieser Ort fällt nicht mit einem Ort im physisch intendierten Raum zusammen.

Die zweite Bedeutungsdimension für Programme sind Gegenstände, die bestimmten Regeln folgend als sprachlich konstituiert werden. Die Intendierung der Gegenstände als sprachlich greift dabei auf solche Regeln wie den erlaubten Zeichensatz (z.B. Buchstaben, Zahlen, Satzzeichen, bestimmte Sonderzeichen usw.), die Anordnung der Zeichen (z.B. zweidimensional, in Zeilen) und Lese- und Schreibrichtung zurück. Ein Beispiel für einen solchen Gegenstand wäre ein abgrenzbares Stück Quellcode. In dieser Bedeutung werden Programme als syntaktische Gegenstände (siehe Kapitel 3.2) begriffen: Der konstituierte Gegenstand ist nach bestimmten formalen Regeln aufgebaut, allerdings wird die Bedeutung der Zeichen zunächst ausgeblendet. So kann auch ein Programm in einer unbekannten Programmiersprache als sprachlich wahrgenommen werden, lediglich der Schluss auf die Semantik setzt genauere Kenntnis der Sprache voraus.

Die Bedeutung stellt die dritte Dimension des Begriffs dar (siehe Kapitel 3.3). Diese enthält zunächst die mathematischen Objekte der formalen Semantik von Programmen. Der

74 Uniform Resource Locator, ein typisches Beispiel sind die Zeichenketten, die für das Aufrufen von Websites verwendet werden wie „https://www.tu-darmstadt.de/“ für die Website der TU Darmstadt (abgerufen am 26.6.2018)

formalen Semantik kommt dabei die Besonderheit beweisbarer Eigenschaften zu: Auch wenn die Wahrnehmung der Sprache zunächst empirisch erfolgt, ist die daraus konstituierte Semantik von weiterer Empirie unabhängig. Da die formale Semantik mathematisch gefasst wird, können – zum Beispiel im Rahmen komplexitätstheoretischer Überlegungen – ihre Eigenschaften mit mathematischen Methoden analysiert werden. Allerdings existiert nicht zu jeder Programmiersprache eine formale Sprachsemantik. Da Programmen auch eine Bedeutung zukommt, wenn für sie keine formale Semantik definiert ist, wird die Bedeutung in dieser Arbeit als Verhalten einer abstrakten Maschine aufgefasst, als maschinelle Semantik (siehe Kapitel 3.3.1).

Neben der maschinellen Semantik können Programme auch als natürlichsprachliche Semantik (siehe Kapitel 3.3.2) erscheinen. Viele der verwendeten Schlüsselwörter wie „if“, „while“, „class“, „void“, „default“ haben eine natürlichsprachliche Bedeutung, verwendete Bezeichner für Variablen und Methoden werden üblicherweise so gewählt, dass eine über die Formalsprache hinausgehende Bedeutung daraus erschlossen werden kann. Und schließlich werden in vielen Programmiersprachen Kommentare im Code ermöglicht, die in natürlicher Sprache weitere Informationen vermitteln. Insbesondere beim Lesen und Verstehen von Programmen kann die natürlichsprachliche Semantik eine große Rolle spielen.

Als vierte Bedeutungsdimension können Programme als Gegenstände konstituiert werden, die ausschließlich in ihrem Zusammenhang und den Wechselwirkungen mit anderen Gegenständen erscheinen. Das sind zunächst alle Programme, die in ihrer Laufzeit konstituiert werden. Ein Programm kann nicht ohne irgendeine Art von Rechner laufen, auf dem es ausgeführt wird. In den allermeisten Fällen sind die Voraussetzungen noch umfangreicher: Von Betriebssystemen über Treiber, Laufzeitbibliotheken, virtuelle Maschinen, Patchserver bis hin zur Netzwerkinfrastruktur und der Stromversorgung kann das Programm als laufend nur eingebettet in eine Vielzahl weiterer technischer Systeme gedacht werden. Die Konstitution des Programms erfolgt also als eingebetteter Gegenstand (siehe Kapitel 3.4).

Neben den Programmen zur Laufzeit (siehe Kapitel 3.4.1) – also Rechenprozessen – umfasst die Vorstellung der Einbettung noch eine weitere Bedeutung. In vielen Fällen werden Programme nicht nur technisch, sondern auch gesellschaftlich eingebettet betrachtet. Die Aussage, einen bestimmten Instant-Messenger oder einen Client für ein soziales Netzwerk zu nutzen, ist neben einer technischen Aussage über die Verwendung eines bestimmten Programms in erster Linie eine Aussage darüber, mit wem und in welcher Art und Weise kommuniziert werden kann. Wenn eine Organisation eine bestimmte Verwaltungssoftware nutzt, bedeutet das, dass das Programm in zahlreiche Prozesse und Strukturen der Organisation eingebunden ist. Ein Campus-Management-System wird zur Laufzeit nicht als auf einem von der Umwelt abgekapselten Laptop konstituiert, sondern als in zahlreiche Abläufe an der Hochschule, in viele technische Systeme und in das Handeln einer großen Anzahl von Menschen eingebunden aufgefasst. Diese Konstitution des Programms als struktureller Bezugspunkt (siehe Kapitel 3.4.2) setzt das Programm in menschliche Handlungszusammenhänge und erfasst sowohl die Ausrichtung des Handelns am Programm als auch das Verhalten des Programms, zum Beispiel als Reaktion auf Eingaben. Die Bedeutungsdimension umfasst dabei auch alle auf das Programm bezogenen Intentionen, die Anwender_innen notwendigerweise mitdenken müssen.

Diese vier Bedeutungsdimensionen des Computerprogramms sind klar unterscheidbar. Die jeweils konstituierten Gegenstände haben unterschiedliche Eigenschaften, beispielsweise kommen einem physischen Gegenstand andere Attribute zu als einem mathematischen Gegenstand. In den nächsten vier Unterkapiteln werden Programme jeweils als Gegenstand in der zugehörigen Bedeutungsdimension detailliert beschrieben. Für eine exaktere Abgrenzung

der verschiedenen Gegenstände können auch innerhalb der Dimensionen Begriffsbedeutungen voneinander getrennt werden, zum Beispiel um zwischen der maschinellen und der natürlichsprachlichen Semantik eines Programms zu unterscheiden. Diese Gliederung des Begriffs ermöglicht eine präzise Eingrenzung der bezeichneten Gegenstände.

Gleichzeitig hängen die Bedeutungsdimensionen jedoch auch voneinander ab und verweisen aufeinander. So kann ein sprachlicher Gegenstand nicht wahrgenommen werden, ohne dass er in physischer Form erfahrbar ist oder gemacht wird. Dieser wechselseitige Verweis der Bedeutungsdimensionen wird hier in Anlehnung an Edmund Husserl und Alfred Schütz (siehe oben, Kapitel 2.2) als Appräsentation, als ein Mitgegenwärtig-Machen aufgefasst. Immer wenn ein Programm in einer der Bedeutungsdimensionen wahrgenommen wird, können die anderen Dimensionen mitgegenwärtig werden.

Diese Appräsentation kann dabei weitestgehend unbestimmt bleiben. Bei der Bedienung eines Fahrkartenautomaten kann mitgegenwärtig sein, dass dieser programmgesteuert ist und dass dieses Programm zumindest im Herstellungsprozess auch in einer sprachlichen Form zugänglich war, ohne dass bekannt ist, in welcher Programmiersprache das Programm geschrieben ist oder welche maschinelle Semantik es genau hat. Beim Schreiben eines Programms können gedachte Anwendungsszenarien (z.B. in Form von Use Cases oder User Stories) eine große Rolle spielen. Dabei wird die Anwendung in unterschiedlichen Abstraktionsebenen gedacht und die Handlungen und beteiligten Menschen (z.B. deren Erwartungen an das Programm, Vorwissen etc.) sind unterschiedlich detailliert appräsentiert.

Eine Appräsentation findet bei den Programmwahrnehmungen nicht zwingend statt. So kann aus Anwendungssicht ein Programm bedient werden, ohne Softwareentwicklungsprozesse oder Programmsemantiken mitzudenken. Eine formale Semantik kann in Quellcode überführt werden, ohne dass Anwendungsszenarien oder die physische Speicherung des Codes auf einem Datenträger beachtet werden. Sobald jedoch bei der Anwendung die Gemachtheit des Programms reflektiert wird, oder bei der Entwicklung des Programms die Anwendung oder auch Auswirkungen des Programms mit bedacht werden sollen, muss eine Appräsentation zwischen den Bedeutungsdimensionen stattfinden. Nur im Sinne des Zusammenhangs durch eine solche Appräsentation kann bei einem selbst geschriebenen und anschließend ausgeführten Programm überhaupt von *einem* Gegenstand ausgegangen werden – beim Schreiben ist die spätere Anwendung, bei der Anwendung das vorherige Schreiben mitgegenwärtig. Ohne diesen wechselseitigen Zusammenhang würde sich das Schreiben auf einen als sprachlich intendierten Gegenstand, die Anwendung auf einen als in ein System eingebettet gedachten Gegenstand beziehen, die vollkommen unterschiedliche Charakteristiken aufweisen.

Neben der wechselseitigen Appräsentation der verschiedenen Bedeutungsdimensionen zeichnet sich der hier entwickelte Programmbegriff noch durch einen weiteren inneren Zusammenhang der bezeichneten Gegenstände aus. Dieser Zusammenhang ist gleichzeitig die Voraussetzung für die Appräsentation wie auch die Voraussetzung für das Programmieren und die Anwendung von Programmen überhaupt. Es geht dabei um die Stabilisierung der Möglichkeiten, Programme in den verschiedenen Bedeutungsdimensionen ineinander zu überführen und wird in Anlehnung an Bruno Latours Akteur-Netzwerk-Theorie Assoziation genannt.

Die Assoziationen zwischen den verschiedenen Bedeutungsdimensionen sind in vielen Fällen voraussetzungsreich und unter hohem Aufwand gesellschaftlich stabilisiert. Die Mittler und Zwischenglieder, die dabei zum Einsatz kommen, reichen von Technologien zum

Schreiben und Lesen von Datenträgern, Maschinen zum Stanzen und Lesen von Lochkarten, ganzen Systemen mit vielen einzelnen Subsystemen (wie z.B. Laptops) bis hin zu Programmiersprachen, natürlicher Sprache und gesellschaftlichen Techniken (wie z.B. formalisierten Softwareentwicklungsprozessen). So wird eine Ansammlung von Zeichen erst dadurch zu einem Java-Programm, dass es eine Spezifikation der Programmiersprache Java gibt, dass diese Spezifikation verfügbar ist, dass in Orientierung an der Spezifikation Compiler geschrieben, gewartet und an neue Hardware und Software angepasst und immer wieder überarbeitet werden. Weiter müssen virtuelle Maschinen für viele verschiedene Systeme entwickelt und überarbeitet werden. Neben der Spezifikation wird die Sprache durch Kurse, Vorlesungen, Tutorials und Bücher stabilisiert, durch die weitere Menschen die Programmiersprache erlernen. In den meisten Fällen kann nur durch Lese- und Anzeigetechniken aus dem physischen Datenträger ein sprachlicher Gegenstand gewonnen werden; die magnetischen Zustände, optischen Oberflächeneigenschaften oder Ladungen auf Floating Gates des Datenträgers werden über mehrere voraussetzungsreiche Interpretationsschichten hinweg als Bits gelesen und über weitere Interpretationsschichten dem Menschen als Schriftzeichen erfahrbar gemacht.

Diese vielfältigen Mittler und Zwischenglieder, welche die verschiedenen Gegenstände miteinander assoziieren, können sich über die Zeit ändern und damit auch die Möglichkeit zur Assoziation ändern. So kann sich eine Programmiersprache oder Standardbibliothek ändern, neue Konstrukte werden Teil der Sprache und alte zunächst „*deprecated*“ und dann nicht mehr unterstützt. Die Abhängigkeit des Programmierens und der Programmanwendung von diesen Mittlern tritt insbesondere dann hervor, wenn die Stabilisierung nicht mehr ausreicht oder fehlschlägt: Wenn für die neue Hardware kein Compiler existiert, wenn die Verbreitung und Vermittlung einer alten Programmiersprache wie COBOL nicht mehr ausreicht, um die darin geschriebenen Systeme zu warten und zu erweitern, wenn ein Datenträger mangels Technik nicht mehr gelesen werden kann.

Nach einer detaillierten Beschreibung dieser Assoziationen (siehe Kapitel 3.5) zwischen den verschiedenen Bedeutungsdimensionen schließt das Kapitel mit einem Exkurs. In diesem werden anhand des hier entwickelten Programmbegriffs Parallelen zwischen Computerprogrammen und musikalischen Kompositionen untersucht. Kompositionen, wie auch Programme, werden dabei als Gegenstände aufgefasst, in denen viele unterschiedliche Teile zusammengesetzt sind und zusammenwirken (siehe Kapitel 3.6).

3.1 Das Programm als räumlich-zeitlicher Gegenstand

3.1.1 Das Programm als physischer Gegenstand

Physische Gegenstände sollen hier Gegenstände sein, die in einem als physisch gedachten Raum zu einer als physisch gedachten Zeit konstituiert werden. Das bedeutet, dass die alltäglichen Vorstellungen von Raum und Zeit denen der Konstitution gleichen. Diese Art von Gegenständen umfasst insbesondere alle unmittelbar mit den Sinnen wahrnehmbaren Gegenstände: Laptops, Datenträger, Lochkarten usw. Sie umfasst auch Gegenstände, die z.B. aufgrund ihrer Größe nicht unmittelbar wahrnehmbar sind, aber trotzdem physisch in Raum und Zeit verortet werden: Floating Gates, Ladungen, Flip Flops und Phosphordotierungen in einem Siliziumkristall können nicht direkt wahrgenommen werden, aber ihnen kann zu einer bestimmten Zeit ein bestimmter Raum zugeschrieben werden. Diese Zuschreibungen sind weder quantenphysikalisch noch relativitätstheoretisch korrekt, die alltägliche Konstitution von Gegenständen in Raum und Zeit ist in dieser Hinsicht vereinfachend.

Die Wahrnehmung physischer Gegenstände stellt die Grundlage zur Konstitution von Computerprogrammen in unserer Erfahrung dar. Damit ist erstens gemeint, dass ein physischer Gegenstand ausschließlich dann als Programm konstituiert werden kann, wenn eine der anderen Bedeutungsdimensionen von Programm appräsentiert ist: Ein Programm kann nicht auf einem Datenträger verortet werden, ohne dass dieses Programm gleichzeitig als sprachlicher, lesbarer oder bearbeitbarer, oder zumindest als unter bestimmten Voraussetzungen ausführbarer, anwendbarer Gegenstand gemeint ist. Die Appräsentation einer anderen Bedeutungsdimension macht den physischen Gegenstand erst zum Programm.

Zweitens ist mit Grundlage gemeint, dass jede direkte Wahrnehmung von Programmen mit den menschlichen Sinnen über physisch gemeinte Gegenstände stattfindet. Das Lesen eines Quellcodes bezieht sich zwar auf den sprachlich konstituierten Gegenstand, die direkt wahrgenommenen Zeichen, sei es auf Papier oder auf einem Monitor, sind jedoch in Raum und Zeit der alltäglichen Erfahrung verortet. Die physische Konstitution kann zwar in den Hintergrund treten, etwa wenn bei der Ausführung eines Programms die Nutzeroberfläche und Funktionalität in der Anwendung als relevante Gegenstände wahrgenommen werden, aber zumindest mittelbar ist das Programm als physische Konfiguration appräsentiert.

Weiter kann die physische Verortung des Programms in der Gegenstandskonstitution auch in verschiedenen Graden von Unbestimmtheit auftreten. Während bei einem ausgedruckten Quellcode die Schriftzeichen auf dem Papier im Ort bestimmt sind, werden bei einer Verortung eines Programms auf einer Festplatte die genauen Positionen von Magnetzuständen auf dieser Platte nicht reflektiert. Wird bei einem komplexen System wie beispielsweise einem Laptop ein Programm innerhalb des Systems verortet kann über das System abstrahiert werden: Die Verortung „im Laptop“ setzt nicht voraus, dass die physische Position des nichtflüchtigen Speichers bekannt ist, oder auch nur dass die Art des Speichers (z.B. Magnetfestplatte oder SSD) bekannt ist.

Ein scheinbarer Grenzfall der physischen Gegenstände als Grundlage der Programmkonstitution sind Programme, die nur in Gedanken existieren. Ein Beispiel wären Überlegungen, wie eine bestimmte Funktionalität implementiert werden kann, oder wie eine Methode geschrieben werden soll. Die Komplexität solcher Programme, die ohne Hilfe

anderer Gegenstände wie einer integrierten Entwicklungsumgebung oder zumindest Papier erdacht werden können, ist aufgrund der Erinnerungsleistung begrenzt. Trotzdem scheint es der grundsätzlichen Möglichkeit, Programme physisch zu verorten, zu widersprechen. Dieser Grenzfall kann dadurch aufgelöst werden, dass jeder Mensch sich selbst in Raum und Zeit der alltäglichen Erfahrung verortet. Dadurch kommt auch den Gedanken und Ideen, die als innerhalb der Selbstkonstitution gemeint sind, Raum und Zeit zu. In der Zuschreibung ist dadurch das Programm „im Kopf“ oder in den physischen Zuständen des Gehirns verortet. Die physische Konstitution des Programms kann also auch hier grundsätzlich appräsentiert werden.

Neben den Datenträgern kann in bestimmten Fällen auch eine Ansammlung mehrerer physischer Gegenstände als Programm konstituiert werden. Wird zum Beispiel ein Programm über klassische Vertriebswege im Einzelhandel erworben, so werden mit dem Kauf neben dem Datenträger häufig eine gedruckte Dokumentation, rechtliche Hinweise wie z.B. Lizenzen, Verweise auf weitere Information (z.B. Website, Telefonnummern für den Support), Verpackung und Werbematerialien ausgehändigt. In einem erweiterten Sinne wird manchmal dieses Gesamtpaket als Programm bezeichnet, zum Beispiel bei der Gestaltung wirtschaftlicher Prozesse, die auf den Verkauf von Programmen abzielen. Je nach Kontext werden insbesondere Dokumentation und Instruktionen häufig als zum Programm zugehörig aufgefasst. Die Konstitution von Programmen in wirtschaftlichen und auch rechtlichen Kontexten geht über die physisch gedachten Gegenstände weit hinaus. Für den hier vorgeschlagenen Begriff sind diese Formen der Wahrnehmung als „eingebettet“ (siehe Kapitel 3.4) aufzufassen: Eine Lizenz, welche die Legalität der Nutzung von Programmen behandelt, ist ohne einen umfassenden Kontext von Recht und Rechtsprechung nicht denkbar.

Wenn Programme als physische Gegenstände konstituiert werden, kommen ihnen über Raum und Zeit hinaus auch andere Eigenschaften physischer Gegenstände zu. So haben Datenträger ein Gewicht, wahrgenommene Farbe, Temperatur, Biegsamkeit usw. Analog zu anderen physischen Gegenständen können diese Eigenschaften sich ändern oder geändert werden. So bleibt eine Diskette, die beschriftet wird, eine Diskette. War vor dem Beschriften ein Programm auf dieser Diskette gespeichert, ist dies bei umsichtigem Vorgehen auch anschließend der Fall. Das als gespeichert appräsentierte Programm hängt nicht von der Beschriftung ab, sondern von den Handlungsmöglichkeiten, die sich aus dem technisch ermöglichten Lesen der Daten auf der Diskette ergeben. Eine Änderung der physischen Eigenschaften muss also keine Auswirkungen auf die Programmkonstitution haben.

Wird hingegen die Diskette, zum Beispiel aus Unwissenheit, zur Archivierung gelocht und in einem Ordner abgeheftet, ist ein möglicherweise gespeichertes Programm nicht mehr lesbar. Die physischen Eigenschaften des Gegenstandes wurden dabei so weit geändert, dass er die Funktion des Speicherns nicht mehr erfüllt. In dieser Hinsicht können die physischen Gegenstände, die als Programme konstituiert werden, als technische Artefakte begriffen werden: Als Gegenstände, denen eine bestimmte Funktion zugeschrieben wird, die sie nur aufgrund ihrer physischen Struktur erfüllen können.[75] Die Funktion der als physisch konstituierten Programme liegt dabei in den anderen Bedeutungsdimensionen: Der physische Gegenstand speichert sprachliche Gegenstände und macht diese unter Nutzung weiterer Techniken lesbar oder ermöglicht, sie zu bearbeiten, oder aber er speichert ein ausführbares Programm, das unter Nutzung weiterer Techniken angewendet werden kann. Sobald die physische Struktur diese Funktionalität nicht mehr ermöglicht kann das Programm nicht mehr im physischen Gegenstand verortet werden.

75 Vgl. Turner, Raymond: Programming Languages as Technical Artifacts. In: Philosophy & Technology Vol. 27 Iss. 3. 2014. S. 379.

Die Funktion der Speicherung – je nachdem ob sie sich auf ein Programm als sprachlichen Gegenstand oder auf ein ausführbares Programm beziehen – kann durch eine ganze Reihe sehr unterschiedlicher technischer Artefakte erfüllt werden. Neben den möglichen Trägern von Schrift (wie Papier) sind zunächst Datenträger zu nennen. Von Lochkarten über Bänder, Datasetten, Disketten, optischen Datenträgern wie CDs oder DVDs, Magnetfestplatten und SSDs bis hin zu möglicherweise zukünftig verfügbaren Techniken wie biologischen Speichern oder dreidimensionalen optischen Speichern können Programme auf sehr heterogenen Medien verortet werden. Flüchtige Speicher, von Prozessorregistern über Buffer und Caches bis hin zum Hauptspeicher eines PCs, Grafikkartenspeicher und dem internen Speicher eines Videoprojektors stellen eine Weitere Menge von Gegenständen dar, in denen ein Programm oder auch Teile eines Programms zeitweise gespeichert werden.

Die Wahrnehmung und Verwendung physischer Gegenstände als Träger von Programmen ist dabei in vielen Fällen technisch voraussetzungsreich. Die Techniken übersetzen in mehreren Kodierungs- und Dekodierungsstufen die physischen Zustände des Gegenstands in eine menschlich wahrnehmbare oder durch Prozessoren ausführbare Form. So werden die Magnetzustände auf einer Festplatte zunächst durch einen Schreib-/Lesekopf bearbeitet. Alleine dieser Vorgang setzt bereits eine Vielzahl an Techniken voraus, wie etwa miniaturisierte Elektromotoren, ausreichenden Schutz vor Staubpartikeln, Reinräume für die Herstellung und Fertigungsverfahren für die Datenscheiben. Die Messung der Magnetisierung durch den Schreib-/Lesekopf wird dabei in einem ersten Schritt digital interpretiert, also als eine Reihe von Werten, die entweder 0 oder 1 annehmen, den Bits (*binary digits*). Über weitere Techniken der Kodierung und Fehlerkorrektur kann die physische Struktur der Festplatte schließlich als eine Reihe von Bits interpretiert werden, auf die in nummerierten Blöcken zugegriffen werden kann. Für die Übermittlung der Daten zu Hauptspeicher und Prozessor werden weitere Techniken der Übertragung und Kodierung angewendet.

Die nächste Kodierungsebene, das Dateisystem, deklariert nun Teile dieser gewonnenen digitalen Daten als zusammengehörige Information. Dateien, sei es als ausführbare Programme oder als Quellcode, sind durch diesen Interpretationsschritt abgrenzbare Gegenstände innerhalb der Menge an Daten auf einem Träger. Die Konstitution des Programms als physisch gemeinter Gegenstand tritt spätestens ab dieser Abstraktionsebene in den Hintergrund: Dateien werden nicht mehr (nur) physisch verortet, die Dateisysteme selbst spannen einen neuen Raum auf, in dem Gegenstände zu einer bestimmten Zeit an einem bestimmten Ort sind. Diese Art, Gegenstände virtuell-physisch zu konstituieren wird im nächsten Abschnitt (Kapitel 3.1.2) behandelt.

Eine Gemeinsamkeit der Vielzahl unterschiedlicher Datenträger ist, dass die gespeicherte Information ab einer bestimmten Interpretationsschicht diskret, als digitale Information verstanden wird. Dadurch werden Eigenschaften als gleich interpretiert, solange sie in einen bestimmten Bereich fallen. Die Diskretisierung erfolgt dabei sowohl über die Zeit als auch über den Wert. Zeitdiskretisierung bedeutet, zum Beispiel bei der Übertragung von Daten über eine Leitung, dass nicht der gesamte Verlauf der Signale interpretiert wird, sondern stattdessen zu fest vorgegebenen Zeitpunkten Messungen stattfinden. Wertdiskretisierung bedeutet zum Beispiel, dass die Magnetzustände auf einer Platte als Bits, als 0 oder 1 interpretiert werden, auch wenn in den physischen Eigenschaften der Platte ein breiteres Spektrum von Zuständen vorkommt.

Neben der digitalen Disziplin, also der Zusammenfassung unterschiedlicher physischer Gegebenheiten zu den Werten 0 und 1 wird noch eine weitere Form von Techniken verwendet, die unterschiedliche Eigenschaften als gleichwertig interpretieren: Kodierungen mit Fehlerkorrektur. Eine Fehlererkennung bezeichnet zunächst die Speicherung zusätzlicher

Information, die das Gelesene unter bestimmten Umständen als inkonsistent kennzeichnet. Ein einfaches Beispiel hierfür ist ein Paritätsbit, das für eine andere Menge an Bits angibt, ob die Anzahl der Einsen gerade oder ungerade ist. Eine Kodierung mit Fehlerkorrektur ermöglicht nun zusätzlich zum Erkennen der Fehler, diese bereits in den Interpretationen des Lesevorgangs zu korrigieren. Ein Anwendungsbeispiel sind CDs, bei deren Lesen zwar durch kleine Kratzer auf der Oberfläche gespeicherte Bits (auch dauerhaft) falsch ausgelesen werden, die Kodierung in vielen Fällen jedoch eine Korrektur der Fehler ermöglicht. Durch die Abstraktion dieser Kodierung werden also unterschiedliche gespeicherte Informationen gleich behandelt, als gleich ausgelesen. In den Begriffen technischer Artefakte ausgedrückt kann durch die Kodierung die Funktionalität bei einer größeren Veränderung der physischen Struktur beibehalten werden.

Einige der Datenträger lassen durchaus das zusätzliche Festhalten analoger Information zu. So werden die durch den Computer lesbaren Daten auf einer Diskette digital interpretiert – die Beschriftung lässt jedoch alle Freiheiten, die Stifte und Papier auch bieten. Statt geschriebener Sprache kann sich auf dem Klebeetikett auch ein gemaltes Bild befinden. Der Ausdruck eines Quellcodes kann handschriftlich annotiert werden, hier kann analoge Information zusätzlich auf das Papier kommen. Auch die Skizze eines Kontrollflussdiagramms enthält analoge Information, etwa die räumliche Anordnung der Programmblöcke oder der genaue Verlauf der Verbindungslinien, die über die mathematischen Eigenschaften des abgebildeten Graphen hinausgehen.

Diese analoge Information wird bei der Konstitution des Gegenstands als Programm in den meisten Fällen vernachlässigt. Der Grund hierfür liegt im Charakter der durch die Programmkonstitution appräsentierten Gegenstände. Wird das Programm im physischen Gegenstand verortet, weil es unter weiteren Voraussetzungen ausgeführt werden kann, so ist der als relevant konstituierte Teil des Gegenstands die in der Ausführung verwendete Information. Da die Lesetechniken ab einer bestimmten Interpretationsstufe die Information digital verarbeiten, wird die zusätzliche analoge Information – z.B. das Bild auf der Diskette – als irrelevant konstituiert. Ist das Programm dagegen als sprachlicher Gegenstand appräsentiert, so erfolgt die weitere Auseinandersetzung damit unter bestimmten Regeln, im weitesten Sinne Grammatiken, die das innerhalb der Sprache Ausdrückbare strukturieren. So ist der Buchstabe „e“ vielleicht in der handschriftlichen Skizze eines Programms in keinen zwei Vorkommen in exakt gleicher Form auf dem Papier festgehalten. Sobald ich die enthaltene Information jedoch als Programm in einer bestimmten Programmiersprache konstituiere begrenzt dies die unterschiedlichen Formen von „e“ durch den verwendeten Zeichensatz – in den meisten Fällen sogar auf nur eine Form. Ein Beispiel mit zwei Formen wäre z.B. eine Sprache, die zwischen „e“ und „E“ unterscheidet – der Unterschied wird dann festgehalten und auch als Information verarbeitet. Durch die Wahrnehmung des physischen Gegenstands mit der Appräsentation eines syntaktischen Gegenstands kommt der analogen Information eine untergeordnete Rolle zu.

Die Wahrnehmung von Programmen in physischen Gegenständen kann neben der Verfügbarkeit von Techniken zum Lesen und zur Interpretation der Daten zusätzlich von weiterer Information abhängig sein. So kann eine Datei, die ein Programm in einer unbekannten Programmiersprache enthält, unter Umständen gar nicht als Programm konstituiert werden. Insbesondere beim Betrachten von kompilierten Programmen in Maschinencode kann es für Menschen ausgesprochen schwierig sein, die dargestellten Daten als Programm zu erkennen. Ein weiteres Beispiel, das noch vor der Interpretation als Gegenstand in einer bestimmten Sprache ansetzt, ist die Verschlüsselung von Daten.

Sowohl Programme als auch andere Dateien können auf Datenträgern in

verschlüsselter Form gespeichert sein. Neben dem Vorhandensein der Technik zum Entschlüsseln erfordert das technisch ermöglichte Lesen, aber auch die Anwendung, zusätzlich einen Schlüssel. Diese Schlüssel können dabei aus Zahlen oder Zeichenketten bestehen, je nach angewandter Kryptographie kann die Entschlüsselung ohne Zugriff auf den Schlüssel schwierig oder unter realistischen Annahmen (z.B. über die zur Verfügung stehende Zeit) unmöglich sein. Diese Charakterisierung von verschlüsselten Daten trifft auch auf verschlüsselte Programme zu. Trotzdem kann auch unter Unkenntnis des Schlüssels ein Programm als auf einem Datenträger befindlich konstituiert werden. Daraus lässt sich folgern, dass die Verortung auf dem Datenträger zwar eine grundsätzliche Möglichkeit des Lesens oder der Ausführung voraussetzt, aber diese Handlungen nicht notwendigerweise unter den aktuellen Voraussetzungen möglich sein müssen. So kann eine Kiste voller Lochkarten auch dann als Programm verstanden werden, wenn ein Lesegerät für diese Karten nicht unmittelbar verfügbar ist.

Dadurch, dass die Konstitution physischer Gegenstände als Programm von der Appräsentation einer anderen Bedeutungsdimension abhängt, können physische Eigenschaften, die nicht unmittelbar mit der Funktion zusammenhängen, als nebensächlich wahrgenommen werden. Solche Eigenschaften können zum Beispiel Farbe, Gewicht, Festigkeit usw. sein – solange sie sich auf die Funktionalität nicht auswirken. Das Gehäuse eines USB-Speichers kann bemalt werden, ohne dass das appräsentierte Programm sich ändert. Wenn jedoch die Kontakte verschmutzt oder der Microcontroller beschädigt werden, ist die Funktionalität (reparabel oder irreparabel) gestört. Das Programm kann nur im physischen Gegenstand verortet werden, solange die gespeicherte Information grundsätzlich noch rekonstruiert werden kann.

Eine Handlung, die die Rekonstruktion des appräsentierten Programms von einem anderen Gegenstand ermöglicht, ist das Kopieren. Bei diesem wird (technisch unterstützt) die gespeicherte Information von einem Datenträger gelesen und auf einen anderen Datenträger geschrieben. Kopieren ist dabei physisch gedacht: Während das Programm vor dem Vorgang an einem bestimmten Ort, in einem Gegenstand, konstituiert wird, wird es anschließend an zwei verschiedenen Orten, als zwei Gegenstände, konstituiert. Die Gemeinsamkeit dieser beiden Gegenstände ist dabei, dass das appräsentierte Programm als sprachlicher oder ausführbarer Gegenstand dasselbe ist. In der physischen Bedeutungsdimension dagegen sind Orginal und Kopie zwei voneinander unterscheidbare Gegenstände. Dies gilt auch, wenn sich Original und Kopie auf demselben Datenträger befinden. So bleibt zwar die genaue Verortung eines Programms auf den Sektoren einer Magnetfestplatte normalerweise unbestimmt, bei der Kopie wird das Programm jedoch ein zweites mal, auf anderen Sektoren, konstituiert.

Eine Handlung, die die Möglichkeit der Rekonstruktion einschränkt oder negiert, ist das Löschen eines Programms. Diese Handlung kann nur physisch verstanden werden: Vor der physisch gedachten Zeitspanne des Löschens kann das verortete Programm gelesen, bearbeitet oder ausgeführt werden, danach nicht mehr. Das Löschen bezieht sich dabei auf *einen* der konstituierten Gegenstände: Wird ein Programm von einem Datenträger gelöscht, kann eine Kopie von einem anderen Datenträger weiterhin gelesen werden. Auch eine Kopie auf demselben Datenträger wird durch das Löschen nicht berührt. Mit dem Löschvorgang wird also nur eine bestimmte Möglichkeit des Zugangs zum sprachlichen oder ausführbaren Programm beendet. Die Intention kann dabei einerseits sein, dass der gelöschte physische Gegenstand nicht mehr benötigt wird, dass Platz auf dem Datenträger für neue Daten freigemacht werden soll. In manchen Fällen ist die Intention des Löschens jedoch auch, dass grundsätzlich niemand mehr auf diesem Weg die Information lesen können soll, zum Beispiel um sensible Patientendaten in einem Krankenhaus zu schützen. Je nach Dateisystem genügt

es für die erstgenannte Intention zum Beispiel, die Sektoren einer Festplatte als unbeschrieben zu deklarieren – sie werden dann beim erneuten Beschreiben mit anderer Information überschrieben. Für die zweitgenannte Intention kommen Techniken wie das (unter Umständen mehrfache) Überschreiben der betroffenen Sektoren zum Einsatz – oder in manchen Fällen auch die Zerstörung des Datenträgers.

Kopien eines Programms oder von Teilen des Programms können bei der Anwendung (zum Beispiel auf einem Laptop) noch an einer Reihe weiterer Orte aufgefunden werden. So wird ein auf einem nichtflüchtigen Speicher verortetes Programm häufig in den flüchtigen Hauptspeicher des Computers geladen, Teile werden zeitweise in Cache, Buffer usw. gespeichert. So kann ein Programm bei der Ausführung in vielen Fällen physisch mehrfach verortet werden. Die Programmkonstitution erfolgt hier im Zusammenhang mit einer Vielzahl weiterer technischer Artefakte, insbesondere auch anderer Programme. Deshalb wird das Programm in der Ausführung hier nicht als rein physisch konstituiert betrachtet, sondern darüber hinaus als eingebetteter Gegenstand (die Konstitution als eingebetteter Gegenstand wird in Kapitel 3.4 behandelt).

In vielen Betriebssystemen können Programme auf Datenträgern verschoben werden. Dieser Vorgang ist bereits metaphorisch als physisch konstituiert. Bei der Verschiebung eines Programms von einem Datenträger auf einen anderen kann der Vorgang als Kopie mit anschließender Löschung des Originals konstituiert (und auch genau so implementiert) werden. Die Verschiebung auf einem Datenträger dagegen ändert (je nach Betriebssystem) die physische Verortung des Programms auf Sektoren, Speicherzellen etc. (je nach Speichertechnologie) gar nicht. Das Verschieben einer Datei zwischen zwei Verzeichnissen eines Datenträgers kann häufig durch die Änderung weniger Metadaten (Verwaltungsdaten) auf dem Datenträger realisiert werden – lediglich die Referenz auf die Datei wird verschoben. Dieses Beispiel zeigt auf, dass der Ort der Datei – das Verzeichnis – nicht bloß physisch konstituiert wird. Die Verortung im Datenträger ändert sich trotz der physischen Metapher des Verschiebens nicht. Stattdessen wird der Datei (im Spezialfall dem Programm) ein neuer virtueller Ort zugewiesen – ein neues Verzeichnis.

3.1.2 Das Programm als virtuell-physischer Gegenstand

Ein Dateisystem eröffnet einen solchen virtualisierten Raum, in dem einzelnen Dateien ein bestimmter Ort innerhalb von Verzeichnissen zukommt. Vom genauen physischen Ort auf dem Datenträger wird dabei abstrahiert. Der aufgespannte virtuelle Raum ermöglicht, die große Menge an gespeicherter Information zu strukturieren, die Verortung in einer Verzeichnisstruktur das Auffinden der Information.

> „A file is an abstraction mechanism. It provides a way to store information on the disk and read it back later. This must be done in such a way as to shield the user from the details of how and where the information is stored, and how the disks actually work.“[76]

Das Dateisystem abstrahiert unter anderem davon, dass die Dateien auf einem Datenträger in vielen Fällen nicht „zusammenhängend“ gespeichert werden, sondern in einer Vielzahl von Blöcken, die auf den freien Platz des Datenträgers verteilt werden. Diese Fragmentierung von Dateien kann bei manchen Speichertechniken zu langsamerem Lesen und Schreiben führen. Muss z.B. der Schreib-/Lesekopf einer Magnetfestplatte aufgrund der Fragmentierung sehr häufig die Spur wechseln um eine Datei zu lesen dauert der Vorgang signifikant länger. Um diese Performanceeinbußen zu reduzieren defragmentieren manche Betriebssysteme die Dateien, indem sie über mehrere Kopier- und Löschvorgänge den Dateien physisch aufeinander folgende Blöcke zuweisen. Aus Nutzerperspektive ist dies nicht sichtbar: Die Datei bleibt virtuell am selben Ort, im selben Verzeichnis, obwohl sie physisch verschoben wurde.

Bei der Verwendung von SSDs, also großer Mengen an Flash-Speicher, anstelle von Magnetfestplatten, ist die Fragmentierung für Lese- und Schreibgeschwindigkeit nicht relevant. Hier gibt es keinen Schreib-/Lesekopf, der für jeden gelesenen oder geschriebenen Block an der richtigen Stelle positioniert werden muss. Bei SSDs spielt jedoch eine andere Eigenschaft eine wichtige Rolle: Speicherzellen nutzen sich nach häufiger Verwendung ab und können ab einer hardwareabhängigen durchschnittlichen Zahl von Schreibvorgängen nicht mehr zuverlässig benutzt werden. Sowohl mit dem Alter des Datenträgers als auch mit der Anzahl an Schreibvorgängen nimmt die Fehlerhäufigkeit zu[77]. Um die Abnutzung möglichst gering zu halten verteilt der Flash-Speicher-Controller Schreibvorgänge möglichst gleichmäßig über die Speicherzellen. Weiter müssen Speicherzellen, falls sie bereits Daten beinhalten, blockweise überschrieben werden. Aus diesem Grund wird eine weitere Abstraktionsschicht verwendet, um die physischen Positionen in Adressen zu übersetzen[78]. Da diese Vorgänge in der Anwendung von Computern und Programmen transparent sind, im Normalfall nicht sichtbar ablaufen, erscheint die virtuelle Verortung von Dateien stabil, auch wenn sich physische Positionierungen ändern. In der Nutzung sichtbar ist lediglich, dass die Datei physisch auf einem Datenträger gespeichert ist und virtuell an einem Ort der Verzeichnisstruktur positioniert werden kann.

Verzeichnisse sind aus der Perspektive des Betriebssystems schlicht Dateien, welche Information über die im Verzeichnis verorteten Dateien (und damit auch über mögliche

76 Tanenbaum, Andrew S., Bos, Herbert: Modern Operating Systems. Fourth Edition. Pearson, 2015. S. 265.

77 Schroeder, B., Lagisetty, R., Merchant, A.: Flash Reliability in Production: The Expected and the Unexpected. In: Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST ’16). 2016. S. 67-80.

78 Chung, Tae-Sun et al.: A survey of flash translation layer. In: Journal of Systems Architecture Vol. 55 Issue 5-6. 2009. S. 332-343.

Unterverzeichnisse) enthalten. Die genaue Art der Organisation von Verzeichnissen variiert zwischen verschiedenen Betriebssystemen, in vielen aktuell verwendeten Betriebssystemen kann die Verzeichnisstruktur als mathematische Baumstruktur oder als gerichteter azyklischer Graph verstanden werden. Gerichtete azyklische Graphen sind allgemeiner – in diesen kann eine Datei „geteilt" sein, also in mehreren Verzeichnissen gleichzeitig verortet werden, in einer Baumstruktur ist das nicht möglich[79]. Das Beispiel dieser *shared files* illustriert, dass im aufgespannten virtuellen Raum die konstituierten Charakteristiken von Raum und Zeit gegenüber Alltagserfahrungen abweichen können, da hier derselbe Gegenstand an zwei Orten gleichzeitig auftritt.

Der umgekehrte Fall, in dem ein virtuell an einem bestimmten Ort positionierter Gegenstand physisch auf mehrere Datenträger verteilt wird, ist zum Beispiel in *RAID*-Systemen (Redundant Array of Independent Disks) zu beobachten. In RAID-Systemen wird das Dateisystem auf mehrere physische Datenträger verteilt. Hierbei gibt es unterschiedliche Architekturen (RAID-level), welche durch die Verwendung mehrerer Datenträger die Performance verbessern und/oder durch Redundanz eine höhere Ausfallsicherheit bieten. Diese Architekturen unterscheiden sich dabei in Performancegewinn, Redundanz und der Resilienz gegenüber verschiedenen möglichen Formen von Fehlern und Ausfällen (z.B. Ausfall eines Sektors, Ausfall einer ganzen Festplatte etc.)[80]. Physisch kann eine Datei (und damit auch Programme) in manchen der RAID-Architekturen nicht mehr auf einem einzelnen Datenträger verortet werden. Virtuell ist sie allerdings weiterhin an einem bestimmten Ort, in einem bestimmten Verzeichnis auf dem durch RAID verfügbaren logischen Laufwerk, positioniert. Bei Handlungen bezüglich der Dateien tritt also die virtuell-physische Konstitution in den Vordergrund, während die Konstitution physischer Gegenstände (der Festplatten) lediglich appräsentiert ist.

Bis hier bezog sich die Virtualisierung hauptsächlich auf einzelne Dateien. Dateien können als virtuelle Gegenstände aufgefasst werden, deren Ort auf eine Weise konstituiert wird, die nicht deckungsgleich mit der Vorstellung von physischem Raum ist. Bei der Gegenstandskonstitution können jedoch auch mehrere Dateien als einzelner Gegenstand erscheinen. Ein einfaches Beispiel hierfür sind Archivdateien. Archivdateien können zunächst dazu verwendet werden, mehrere Dateien zu einer einzelnen zusammenzufassen. Diese stellt sich sowohl für das Betriebssystem als auch in der Anwendung als einzelne Datei dar, die zusammengefassten Dateien können bei Bedarf wieder extrahiert werden. In vielen Fällen wird noch zusätzliche Funktionalität mit der Archivierung verknüpft, etwa die Komprimierung der Archivdatei, um Speicherplatz zu sparen, oder eine Verschlüsselung, die einen Zugriff erst nach Authentifikation (z.B. durch Kenntnis eines Schlüssels oder eines Kennworts) ermöglicht. Die Konstitution der Archivdatei als einzelner Gegenstand fällt hier mit der technischen Realisierung als einzelne Datei zusammen. Sie fasst dabei den extrahierbaren Inhalt des Archivs als zusammen gehörenden abgegrenzten Gegenstand auf.

Viele Archivierungsprogramme ermöglichen auch ein Aufteilen der Archivdatei in mehrere Einzeldateien (Multi-Volume Archives). Diese Technik kann zum Beispiel dazu verwendet werden, ein Archiv auf mehrere Datenträger zu kopieren, wenn die Archivgröße die Kapazität eines einzelnen Datenträgers übersteigt. In der Wahrnehmung können sowohl die einzelnen Teilarchive (die Dateien) als Gegenstand konstituiert werden als auch das gesamte Archiv. In letzterem Fall stimmt die Gegenstandskonstitution nicht mehr mit der technischen Realisierung als Datei überein, der virtuell-physisch gemeinte Gegenstand ist das gesamte Archiv als inhaltlich zusammengefasste Einheit.

79 Zu Shared Files siehe Tanenbaum, Andrew S., Bos, Herbert: Modern Operating Systems. Fourth Edition. Pearson, S. 290ff.

80 Siehe ebda. S. 371ff.

Programme bestehen in vielen Fällen nicht aus einer einzelnen Datei. Das gilt sowohl für die lesbaren und bearbeitbaren Quellcodedateien als auch für ausführbare Programme. Der Quellcode kann beispielsweise aufgeteilt und – je nach verwendeter Programmiersprache – zusätzlich zum Aufbau durch die vorhandenen Konstrukte wie Klassen, Methoden, Funktionen etc. auch über diese Aufteilung strukturiert sein. Weiter können sich notwendige Teile des Programms in statischen Bibliotheken, Konfigurationsdateien, Makefiles usw. befinden. Bei der Ausführung kann – je nach Anwendungsgebiet – der Zugriff auf verschiedene in Dateien festgehaltene Daten über dynamische Bibliotheken, Logdateien bis hin zu Grafiken und Audiodateien für die Benutzeroberfläche erfolgen. Das Programm wird als zusammengehörender Gegenstand, der aus mehreren Dateien besteht, konstituiert. Die virtuelle Verortung muss dabei nicht zwangsweise anhand des Dateisystems erfolgen. So sind die Dateien häufig verteilt, dynamische Bibliotheken befinden sich je nach Betriebssystem in zentral verwalteten Verzeichnissen, Datendateien können auf externen Datenträgern liegen.

Bei Programmen, die aus mehreren Dateien bestehen, können einzelne Programmteile jeweils in verschiedenen Dateien verortet werden. So wird Klassen, Methoden, Konfigurationsdaten usw. ein virtueller Ort zugeschrieben, die jeweilige Datei, in denen sich dann die Implementierung, Wertsetzung usw. befindet. Die Verweise, z.B. als *include*-Anweisung für den Compiler, können räumlich als Ortsbeschreibungen verstanden werden, an denen sich die betreffenden Teile des Quelltextes befinden. Auch das Öffnen und Schreiben von Dateien zur Laufzeit findet über solche Verweise statt, in denen (abhängig vom Betriebssystem) Pfad und Dateiname angegeben werden. Die Abgrenzung des Programms als Gegenstand kann je nach Intention unterschiedlich ausfallen. So kann zum Beispiel eine verwendete Bibliothek als Teil des Programms konstituiert werden – oder aber als separater Gegenstand, der für eine Kompilierung (*statisches Binden*) oder für eine spätere Ausführung (*dynamisches Binden*) des Programms notwendigerweise vorhanden sein muss. Die Abgrenzung des Gegenstands kann in diesem Beispiel davon abhängen, ob die Bibliothek im Rahmen der Entwicklung mit geschrieben wurde oder von anderen Projekten oder Teams übernommen wurde. Es ist für die Abgrenzung relevant, wie spezifisch die Bibliothek auf das Programm zugeschnitten ist, in das sie eingebunden wird, ob sie mit anderen Programmen geteilt wird usw. Durch den modularen Aufbau von Programmen, für den Bibliotheken nur ein Beispiel sind, kann die Abgrenzung in der Gegenstandskonstitution von Fall zu Fall variieren.

Eine besondere Form virtuell-physisch konstituierter Programme sind verteilte Programme. Diese Programme laufen gleichzeitig auf mehreren vollständigen Computern, einem verteilten System, und realisieren Anwendungen, die auf die Vielzahl von Recheneinheiten und Nutzerschnittstellen eines solchen Systems zurückgreifen können. Sie können, in der Systemperspektive, als einzelner Gegenstand konstituiert werden. Beispiele für Architekturen verteilter Systeme sind Client-Server-Systeme oder Peer-to-Peer Netzwerke. Ein verteiltes Programm kann als Gegenstand virtuell auf allen beteiligten Computern gleichzeitig oder „im Netzwerk“ verortet werden. Der Austausch von Daten zwischen den Elementen eines verteilten Systems findet dabei über spezifische Protokolle statt, die den virtuellen Raum des Netzwerks mit einem eigenen System der Ortszuschreibung versehen.

Als bekanntes Beispiel eines sehr großen verteilten Systems soll hier das World Wide Web (www) zur Illustration herangezogen werden. Es verwirklicht eine Client-Server-Architektur: Webserver stellen auf Anfrage von Clients (z.B. Browsern) über ein festgelegtes Protokoll (http) Webseiten zur Verfügung, deren Sprache (html) der Client dann interpretiert und z.B. für Menschen lesbar darstellt. Im System selbst wird eine spezifische Form der Orientierung im Raum festgelegt, die Verortung von Seiten und anderen Dateien über URLs. Der dabei aufgespannte Raum ist virtuell: Eine URL kann gleich bleiben, auch wenn die

dadurch bezeichnete Datei auf einen anderen Server transferiert wird. Umgekehrt kann auf den gleichen Server über unterschiedliche URLs zugegriffen werden. Diese Virtualisierung von Raum wird dadurch ermöglicht, dass ein weiteres verteiltes System (das Domain Name System, DNS) die Übersetzung des in der URL enthaltenen Hostnamens in eine IP-Adresse ermöglicht. Das Internet Protocol (IP) beschreibt hierbei den Ablauf der Kommunikation zwischen zwei auf diese Art adressierbaren Computern.

Kommunikationsprotokolle wie das genannte Internet Protocol können als in mehreren Schichten aufeinander aufbauend modelliert werden. Eine häufig verwendete Modellierung der Kommunikation in Netzwerken erfolgt durch das Open Systems Interconnection Model (OSI Model), in dem sieben aufeinander aufbauende Schichten von Netzwerkprotokollen beschrieben werden[81]. Für diese Arbeit ist dabei relevant, dass in den aufeinander aufbauenden Schichten auch unterschiedliche Virtualisierungen von Raum stattfinden können. So stellt die Konstitution von Programmen als Dateien in Dateisystemen bereits unabhängig von Netzwerkprotokollen eine Abstraktion von physischem Raum dar. MAC (Medium Access Control) weist in der zweiten Schicht des OSI-Modells einzelnen Netzwerkinterfaces eigene Adressen zu und verortet Geräte darüber. IP verwendet einen eigenen Adressraum in der dritten Schicht, das Transmission Control Protocol (TCP) fügt diesem noch einen Raum von Ports hinzu, mit dem mehrere nebenläufige Verbindungen zwischen zwei Endpunkten unterschieden werden. Jede dieser Adressierungsformen spannt einen eigenen Raum auf, in dem unterschiedliche Geräte und Anwendungen verortet werden können.

Ein weiteres Beispiel für die Virtualisierung von Raum bei Computerprogrammen ist die Speicherverwaltung. In den meisten heutigen Betriebssystem stellt sich der Hauptspeicher eines Computers für Programme als virtueller Adressraum dar: Das Betriebssystem übersetzt die physische Adressierung von Speicher in einen virtuellen Raum, der sich jedem Programm gleich darstellt. Bei Zugriffen auf den Speicher wird jeweils die virtuelle Adresse, auf die das Programm schreiben oder die es lesen soll, vom Betriebssystem in Zusammenarbeit mit der Hardware in physische Adressen übersetzt. Dadurch wird eine höhere Systemstabilität und Sicherheit erreicht, da Programme so im Normalfall andere Programme (und auch das Betriebssystem) nicht zum Absturz bringen können, indem verwendeter Speicher überschrieben wird. Weiter sind im Hauptspeicher liegende Daten dadurch gegenüber einem ungewollten lesenden Zugriff geschützt[82]. Diese Technik ist insbesondere (aber nicht nur) bei der gleichzeitigen Verwendung mehrerer Programme, beim Multitasking, relevant. Der virtualisierte Speicher kann dabei den physisch vorhandenen Hauptspeicher in der Größe übertreffen, indem Teile auf einen größeren, langsameren Speicher, zum Beispiel eine Festplatte, ausgelagert werden.

In der Anwendung kann ein Programm virtuell über die grafische Benutzeroberfläche des Betriebssystems verortet werden, zum Beispiel wenn der Ort einer Smartphone-Applikation lediglich durch den virtuellen Ort der Verknüpfung, die die Applikation startet, gedacht wird. Dateisystem und Benutzeroberfläche stellen hier zwei unterschiedliche Formen der Abstraktion und Virtualisierung von Raum dar, die auf unterschiedliche Zugänge in der Anwendung abzielen. Dateisysteme virtualisieren den Raum als möglichst einfache Struktur, als Baum oder gerichteten azyklischen Graphen, in dem Gegenstände über eindeutige Namen auffindbar sind. Die relevante Information zur Verortung von Gegenständen ist hier der Pfad, z.B. vom Wurzelverzeichnis in das Unterverzeichnis mit der gesuchten Datei. Grafische

81 Für eine detaillierte Beschreibung siehe Tanenbaum, Andrew S., Wetherall, David J.: Computer Networks. Fifth Edition. Prentice Hall, 2010. S. 41ff.

82 Siehe Tanenbaum, Andrew S., Bos, Herbert: Modern Operating Systems. Fourth Edition. Pearson, 2015. S. 185ff.

Benutzeroberflächen, zum Beispiel Fenstersysteme, virtualisieren nahe an der intuitiven Vorstellung von physischem Raum. Ein Ort ist hier eine Position auf der zweidimensionalen Fläche, metaphorisch als Desktop bezeichnet und der zweidimensionalen Fläche eines Schreibtisches nachempfunden. Für den Begriff des Computerprogramms ist relevant, dass bei der Auseinandersetzung mit Computern und Programmen häufig mehrere Formen der Virtualisierung gleichzeitig stattfinden und sich überlagern können. So kann in einem Fenstersystem, das ein Dateisystem darstellt (z.B. durch einen Dateimanager), gleichzeitig die Graphenstruktur und die zweidimensionale Positionierung auf einem Desktop oder in Fenstern präsent sein. Die Gegenstandskonstitution kann dabei je nach Intention die eine oder die andere Form der Virtualisierung von Raum betonen.

Neben dem Raum wird im Zusammenhang mit Computerprogrammen auch die Zeit virtualisiert. Analog zum Raum bedeutet dies, dass eine eigene Zeitlichkeit mit einer eigenen Bedeutung von „vorher“, „nachher“ oder auch „gleichzeitig“ konstruiert werden kann. Ein Beispiel für die Virtualisierung von Zeit findet sich bei der Ausführung mehrerer Programme auf einem Computer, beim Multitasking. Während bei echten Mehrkernprozessoren zwei Programme auch physisch gleichzeitig ausgeführt werden können simuliert in vielen Fällen das Betriebssystem eine Gleichzeitigkeit, die physisch nicht stattfindet.

> „In any multiprogramming system, the CPU switches from process to process quickly, running each for tens or hundreds of milliseconds. While, strictly speaking, at any one instant the CPU is running only one process, in the course of 1 second it may work on several of them, giving the illusion of parallelism. Sometimes people speak of **pseudoparallelism** in this context, to contrast it with the true hardware parallelism of **multiprocessor** systems (which have two or more CPUs sharing the same physical memory).“[83]

Dieser Pseudoparallelismus kann bei der Verwendung mehrerer Threads, also einzelner Ausführungsstränge eines Programms die zumindest teilweise unabhängig voneinander sind, auch innerhalb eines einzelnen Programms stattfinden. Der Prozessor arbeitet dabei immer nur an einem Ausführungsstrang, physisch können mehrere Threads und Prozesse nicht von einem Prozessorkern gleichzeitig bearbeitet werden. Durch den häufigen Wechsel der bearbeiteten Threads erscheint hier eine virtuelle Gleichzeitigkeit, die auch in der Konstitution von Programmen übernommen wird. Auf Mehrkernprozessoren dagegen kann das Betriebssystem dem einen Programm mehrere Kerne zur Verfügung stellen, hier gibt es neben der virtuellen auch tatsächliche Gleichzeitigkeit. Dabei kann eine solche virtuelle Gleichzeitigkeit (und auch tatsächliche Gleichzeitigkeit bei Mehrkernprozessoren), insbesondere innerhalb eines Programms, eigene Schwierigkeiten mit sich bringen: Wenn zwei nebenläufige Threads versuchen, auf die gleichen Ressourcen zuzugreifen, oder ein Thread auf ein Ergebnis eines anderen Threads angewiesen ist. Die Zuteilung von Ressourcen und insbesondere Rechenzeit des Prozessors zu den einzelnen Threads und Prozessen ist ein komplexes Problem zu dessen Lösung viele unterschiedliche *Scheduling*-Algorithmen existieren[84].

Eine beim Programmieren häufig verwendete Abstraktion über Zeit sind Versionsnummern. Eine höhere Versionsnummer drückt dabei aus, dass die bezeichnete Version „neuer“ ist, vor einer kürzeren Zeit erstellt wurde. Die Nummer abstrahiert dabei von der physisch konstituierten Zeit, in der die jeweilige Programmversion tatsächlich fertiggestellt wurde. Zur Nummerierung verschiedener Versionen gibt es dabei

83 Tanenbaum, Andrew S., Bos, Herbert: Modern Operating Systems. Fourth Edition. Pearson, 2015. S. 86.

84 Für einen Überblick siehe ebda. S. 149ff.

unterschiedliche Systeme. Manche referenzieren andere Zeitdarstellungen, zum Beispiel nummeriert die Linux-Distribution Ubuntu die veröffentlichten Versionen anhand des verbreitet verwendeten gregorianischen Kalenders (Version 18.04 erschien im April 2018). Eine Gemeinsamkeit vieler Systeme, nach denen Versionsnummern vergeben werden, ist dabei, dass zwischen verschieden großen Schritten bei der Weiterentwicklung des Programms unterschieden wird: Eine Änderung in der ersten aufsteigenden Zahl deutet „große“ Schritte an, z.B. eine umfassende Erweiterung der Funktionalität, während die zweite Zahl vielleicht nur auf kleinere Verbesserungen und die dritte lediglich auf dringend notwendige Fehlerkorrekturen (sogenannte hotfixes) hindeuten. Über die Versionsnummern kann erkannt werden, welche Version früher und welche später veröffentlicht wurde, in vielen Fällen jedoch nicht, wie der genaue zeitliche Verlauf war (mit Ausnahme solcher Versionierungssysteme wie Ubuntu es verwendet). Das „vorher“ und „nachher“ der Versionen ist also virtuell und nicht physisch gegeben. Eine weitergehende Auseinandersetzung mit Zeit in Computerprogrammen findet sich auch in Kapitel 4.4 dieser Arbeit. Die Verwendung bestimmter Systeme zur Versionierung ist eng verknüpft mit dem jeweiligen Prozess, mit dem Programme entwickelt werden. Eine neue Version sagt dabei aus, dass am Quellcode des Programms etwas geändert wurde. Diese Änderungen können auch rein textlich sein, ohne dass die Funktionalität des Programms oder die Semantik sich ändern. Änderungen zwischen Versionen eines Quellcodes sind immer (mindestens) syntaktischer Natur. Wie der Quellcode als syntaktischer Gegenstand konstituiert wird, wird im nächste Unterkapitel betrachtet.

3.2 Das Programm als syntaktischer Gegenstand

Insbesondere bei der Programmierung, der Fehlersuche, der Analyse von Programmen und dem Erlernen von Programmiersprachen begegnen uns Computerprogramme als Quellcode. Quellcode in seiner Vielzahl unterschiedlicher Formen wird dabei als sprachlicher Gegenstand konstituiert, der nach bestimmten Regeln aufgebaut ist. Aus diesem Grund wird diese Art von Gegenständen hier als syntaktisch verstanden. Werden Programme als syntaktische Gegenstände konstituiert, tritt der physische Gegenstand in den Hintergrund: Bei der Arbeit an Quellcode kann, zumindest in weiten Teilen, davon abstrahiert werden, wie dieser gespeichert ist. Der gleiche Quellcode kann auf Papier, an einer Tafel, in einer Entwicklungsumgebung oder mit einem einfachen Texteditor bearbeitet werden. Die Bearbeitung am Computer stellt dabei zwar andere Werkzeuge zur Verfügung als Papier und Bleistift, der konstituierte sprachliche Gegenstand bleibt dagegen gleich.

Für die Konstitution als Quellcode ist es notwendig, dass der Gegenstand von Computern verarbeitet werden kann. Daher sind die hier behandelten Gegenstände nach den Regeln aufgebaut, die eine Speicherung und Verarbeitung von Text an Computern möglich machen. Einfache Textdokumente sind dabei als endliche Reihe von Zeichen konstruiert. Es gibt ein Anfangs- und ein Endzeichen, jedes weitere Zeichen hat je genau einen Vorgänger und Nachfolger und es gibt eine Leserichtung vom Anfang zum Ende. Im Gegensatz zum freien Schreiben auf Papier ist die Position einzelner Zeichen alleine durch die Position innerhalb der Reihe gegeben: Wird Text auf Papier als Programm konstituiert, so kommt einer Positionierung einzelner Zeichen über die Verortung in der Reihe hinaus keine Bedeutung zu. In einigen Programmiersprachen (z.B. Python – hier wird das Programm über Einrückungen strukturiert) enthält die Positionierung von Zeichen innerhalb einer Zeile Information, die auch maschinell verarbeitet wird. In der Programmiersprache Fortran kann eine Formatierung verwendet werden, die den Spalten Bedeutungen zuschreibt – dieses Format war auf die Verwendung von Lochkarten zur Speicherung von Programmen ausgelegt. Trotzdem können diese Programme als Reihe von Zeichen verstanden werden, indem Zeilenumbrüche auch als solche Zeichen innerhalb der Reihe aufgefasst werden.

Die Menge unterscheidbarer Zeichen ist dabei durch einen Zeichensatz gegeben. Solche Zeichensätze enthalten zum Beispiel Buchstaben, Zahlen, Sonderzeichen und Steuerungszeichen. Durch eine Kodierung kann jedes Zeichen aus dem Zeichensatz in eine natürliche Zahl übertragen werden, die (meist als Byte oder Folge weniger Bytes) vom Computer verarbeitet wird. Historisch sind dabei ASCII-Zeichensatz und -codierung von Bedeutung (American Standard Code for Information Interchange), viele gegenwärtig verwendeten Programmiersprachen und -werkzeuge lassen die Verwendung einer Teilmenge des Unicode-Zeichensatzes zu und greifen auf eine entsprechende Kodierung (z.B. UTF-8) zurück. Unicode ist ein wachsender Zeichensatz, der regelmäßig um neue Zeichen erweitert wird, mit dem Ziel, jedes von Menschen verwendete Schriftzeichen zu erfassen[85]. Da der ASCII-Zeichensatz deutlich eingeschränkter ist als Unicode ermöglicht die Umstellung z.B. die Verwendung von Umlauten in Programmen. Die Zeichen sind dabei durch die Kodierung eindeutig und voneinander unterscheidbar.

Eine eindimensionale Reihe von Zeichen ist für Menschen schwer lesbar. Text an Computern wird daher entweder durch automatische Zeilenumbrüche zweidimensional dargestellt, oder der Text selbst enthält Steuerzeichen, die einen manuellen Zeilenumbruch

85 „The Unicode Standard provides a unique number for every character, no matter what platform, device, application or language.“ https://www.unicode.org/standard/WhatIsUnicode.html (abgerufen am 31.07.2018).

veranlassen. Während die Mehrzahl an Zeilenumbrüchen in Fließtext automatisch erfolgt, indem die verarbeitenden oder darstellenden Programme die Umbrüche nach einem Algorithmus einfügen, der gute Lesbarkeit erzeugen soll, kommt in Programmen den manuellen Umbrüchen eine größere Bedeutung zu. Dies liegt daran, dass sich Quellcode je nach Programmiersprache anhand der Unterteilung in Instruktionen, Methoden usw. strukturieren lässt. So ist es z.B. in vielen Programmiersprachen üblich, jede Instruktion in einer neuen Zeile zu beginnen. Dafür sind in der Zeichenkodierung bestimmte Steuerungszeichen festgelegt, die solche manuellen Zeilenumbrüche markieren. Im Ergebnis werden Computerprogramme als syntaktische Gegenstände in den meisten Fällen zweidimensional konstituiert, obwohl die interne Repräsentation im Computer eine eindimensionale Reihe von Zeichen ist. Für Menschen ist es in den meisten Fällen leichter, sich anhand von Zeilennummern zu orientieren und Struktur über eine zweidimensionale Anordnung zu erfassen, anstatt mit der eindimensionalen Repräsentation zu arbeiten. Die zweidimensionale Konstitution von Text ist außerdem deutlich näher an der Verwendung von Papier, Tafeln, Schildern usw. welche auch als zweidimensionale Lese- und Schreibfläche erscheinen.

Wie diese Strukturierung nun im Detail aussieht ist von der Syntax jeder einzelnen Programmiersprache abhängig. Da diese Sprachen sehr heterogen sind, sind die meisten folgenden Beschreibungen zur Syntax nicht allgemeingültig, zu fast jedem Detail lassen sich einzelne Sprachen als Gegenbeispiel aufführen. Trotzdem ist eine Auseinandersetzung mit weit verbreiteten Merkmalen der Syntax für diese Arbeit notwendig, um die Gegenstandskonstitution zu analysieren. Zum Beispiel gibt es Programmiersprachen, die keine manuellen Zeilenumbrüche zulassen, und auch solche, die kaum als zweidimensionaler Text konstituiert werden können, wie etwa Maschinensprache. Die Merkmale sind jedoch in den höheren Programmiersprachen so weit verbreitet, dass sie in den Erwartungen, wie Programme sich sprachlich darstellen, fest verankert sind. Trotz Kontingenz bestimmen diese Erwartungen die Gegenstandskonstitution mit. Daher lohnt es sich, trotz fehlender Allgemeingültigkeit detaillierter auf syntaktische Merkmale einzugehen.

Zeilenumbrüche, Leerzeichen, Einrückungen usw. werden beim Programmieren als *Whitespace* oder Leerraum bezeichnet, da dadurch keine anzeigbaren Zeichen beschrieben werden, sondern die Anordnung der anderen angezeigten Zeichen verändert wird. Die verschiedenen Whitespace-Zeichen, insbesondere Leerzeichen, werden in vielen Sprachen dazu verwendet, Wörter voneinander zu trennen. So wird das Schlüsselwort *while*, das eine Schleife – also Code, der mehrmals hintereinander ausgeführt werden kann – unter anderem dadurch als while erfasst, dass vor und nach den Buchstaben Whitespace verwendet wird. Über die Funktion der Trennung hinaus hat Whitespace in einigen Sprachen keine Bedeutung, wodurch er relativ frei zur Anordnung und Strukturierung des Programms verwendet werden kann, um eine gute Lesbarkeit zu erreichen. Zum Beispiel werden die Inhalte einer while-Schleife häufig eingerückt, um ihren durch die Schleife gegebenen Zusammenhang zu kennzeichnen.

Der Umgang mit Whitespace gehört zur sprachspezifischen Syntax. Ein weiteres Element sprachspezifischer Syntax, das sich in fast allen höheren Programmiersprachen findet, sind Trennzeichen. Häufig kommen eine ganze Reihe unterschiedlicher Trennzeichen zur Anwendung, um verschiedene Abgrenzungen von Zeichenmengen und ihren Bezug zueinander darzustellen. So beenden z.B. in Java Semikolons (;) Anweisungen, Kommata trennen mehrere Variablen in Methodendeklarationen und Aufrufen voneinander, Punkte trennen Objekte und Methoden. Einzelne Trennzeichen erfüllen häufig mehrere Funktionen je nach Kontext, z.B. trennen Punkte sowohl Paketnamen von Klassen als auch Klassen von

deren Methoden. In vielen Sprachen werden auch verschieden Arten von Klammern verwendet, die jeweils paarweise eine bestimmte Art von Zusammenhang der eingeschlossenen Zeichen kennzeichnen. Analog zu mathematischen Termen können Klammern geschachtelt verwendet werden.

Die sprachspezifische Syntax gibt an, meist mit Rückgriff auf mehrere Ebenen, wie ein gültiges Programm in einer bestimmten Sprache aufgebaut ist. Auf der ersten Ebene wird beispielsweise definiert, wie ein Wort oder *Token* aussieht, etwa durch die Definition von Trennzeichen. Die zweite Ebene würde dann eine formale Grammatik darstellen, die die Formen gültiger Wortreihen festlegt. Eine dritte Ebene könnte bereits die Verbindung zur maschinellen Bedeutung darstellen, z.B. indem definiert wird, wie genau Methodenaufrufe mit dem Code der Methode verknüpft sind. Viele Compiler, die aus Quellcode ausführbare Programme erzeugen, arbeiten mit diesen drei Ebenen, als lexikalischer, syntaktischer und semantischer Analyse. Die erste und zweite Ebene werden dabei häufig zu einem *Parser* zusammengefasst.

Die Konstitution des Computerprogramms als sprachlicher Gegenstand orientiert sich dabei stark an der Struktur, die durch die syntaktischen Regeln vorgegeben wird. So werden die durch Trennzeichen abgegrenzten Wörter auch beim Lesen des Programms als separate Teile wahrgenommen. Die durch die Grammatik definierten Phrasen können einzeln konstituiert und dann in ihrem Zusammenhang gesehen werden. Bezeichner werden auch in der Wahrnehmung semantisch analysiert: So ist die Erwartung, dass durch das zweimalige Auftreten eines Variablennamens dieselbe Variable gemeint ist, gleiche Methodennamen die gleiche Methode bezeichnen[86]. Die Form, in der Programme syntaktisch strukturiert sind, schlägt sich in der Konstitution als Programm nieder.

Viele Programmiersprachen lassen syntaktisch gekennzeichnete Kommentare zu. Diese werden meist durch spezifisch dafür verwendete Trennzeichen vom Rest des Quellcodes abgetrennt. Compiler entfernen diese Kommentare während der lexikalischen Analyse, sie haben für die weitere maschinelle Verarbeitung und die spätere Ausführung keine Bedeutung. Ihr Zweck liegt in der Bereitstellung von Informationen für Menschen, die das Programm lesen. Dabei kann sowohl Informationsaustausch in Programmierteams stattfinden als auch Notizen für die weitere Arbeit festgehalten werden, wenn nur ein Mensch am Programm arbeitet. Bei Open Source-Programmen mit frei verfügbarem Quellcode wird diese meist natürlichsprachlich formulierte Information möglicherweise einer sehr großen Gruppe zur Verfügung gestellt. Die in Kommentaren dargestellte Information kann dabei von Information zur Erstellung des Quellcodes (zum Beispiel wer ihn wann geschrieben hat, Informationen zur Lizenz, Verweise auf weitere Information wie websites) über Erläuterungen und Begründungen zu einzelnen Teilen, Hinweisen zur Verwendung (z.B. bei Bibliotheken), Hinweisen zu Fehlern und zu notwendiger weiterer Arbeit bis zu spezifischer Domäneninformation aus dem Anwendungsgebiet des Programms gehen. Normative Einschätzungen von Kommentaren sind vielfältig, es gibt mehrere Richtlinien und Konventionen dazu, wie Quellcode kommentiert sein *sollte*, und diese sind nicht immer widerspruchsfrei. Diese Diskussionen lassen sich im breiteren Feld der ästhetischen Bewertung von Computerprogrammen verorten, welches im Ausblick am Ende dieser Arbeit aufgegriffen wird.

86 Eine bemerkenswerte Ausnahme stellen Overloading und Overriding dar, eine Programmiertechnik in der objektorientierten Programmierung, die mehrere Methoden unter demselben Namen verwendet. Während dies hier nur angemerkt werden kann, wäre es eine interessante Fragestellung für eine nachfolgende Untersuchung, wie bestimmte Programmierparadigmen wie objektorientiertes Programmieren die Gegenstandskonstitution beeinflussen. Ein weiteres Beispiel wäre die Gegenüberstellung von Methodenaufrufen mit möglichen Seiteneffekten mit Funktionsaufrufen in funktionaler Programmierung – und wie diese unterschiedlichen Herangehensweisen die Wahrnehmung von Unterprogrammmen beeinflussen.

In einigen Kontexten werden solche Kommentare noch mit weiteren Funktionen versehen. Diese Funktionen reichen dabei von Anweisungen für den Compiler über eingebetteten Quellcode in anderen Programmiersprachen und Spracherweiterungen bis zu Informationen für weitere Programmierwerkzeuge. Zum Beispiel werden Spracherweiterungen für parallele Programmierung manchmal als Kommentare in der eigentlich verwendeten Programmiersprache ausgedrückt – und können so sowohl parallel als auch sequentiell übersetzt werden, ohne den Quellcode zu ändern. Kommentare in dieser Funktion sollen hier als „unecht“ bezeichnet werden: In der Gegenstandskonstitution stellen sie gerade keinen Kommentar dar, sondern eine Anweisung für das jeweilige Werkzeug. Im Gegensatz zu echten Kommentaren bezwecken sie Unterschiede in der weiteren maschinellen Verarbeitung des Codes. Ist ein Programm fehlerhaft, so kann der Fehler sehr wohl in den unechten Kommentaren liegen (im Beispiel der parallelen Programmierung ist dies nicht einmal unwahrscheinlich) – ein echter Kommentar dagegen verändert das Verhalten zur Laufzeit gar nicht, im Übersetzungsprozess wird er üblicherweise während der lexikalischen Analyse ignoriert und nicht in die weiterverarbeiteten Tokens überführt.

Höhere Programmiersprachen ermöglichen es, Programme in Deklarationen, Anweisungen, Methodenbeschreibungen, Klassenbeschreibungen usw. auszudrücken. Die einzelnen Elemente hängen dabei von der für die jeweilige Programmiersprache definierten Syntax ab. Die Struktur, in der das Programm wahrgenommen wird, ist dabei an den Strukturierungselementen orientiert, die auch bei der maschinellen Verarbeitung verwendet werden. Die Funktion der höheren Programmiersprachen ist dabei zweiteilig: Erstens soll der Quellcode maschinell verarbeitet werden können, insbesondere soll durch Compiler oder Interpreter der Code ausführbar sein. Ein Interpreter führt dabei direkt Code der höheren Sprache aus, ein Compiler übersetzt diesen in eine andere Sprache. Diese andere Sprache ist in vielen Fällen Maschinensprache, die wiederum direkt ausgeführt werden kann. Zweitens sollen die höheren Programmiersprachen ermöglichen, den Code in einer durch Menschen relativ leicht lesbaren und schreibbaren Form festzuhalten.

Maschinensprache, also direkt durch Prozessoren ausführbare Instruktionen, und höhere Programmiersprachen unterscheiden sich dabei in mehreren Aspekten. Zunächst ermöglicht es eine höhere Programmiersprache, Programme relativ unabhängig von der verwendeten Hardware und teilweise unabhängig von weiterer verwendeter Software zu formulieren. So kann das gleiche in einer höheren Sprache formulierte Programm nach Übersetzung auf Prozessoren mit unterschiedlichem Befehlssatz und unterschiedlicher Anzahl und Art der Register usw. laufen, und in manchen Fällen zusammen mit verschiedenen Betriebssystemen, Treibern, Hilfsprogrammen, Bibliotheken usw. funktionieren. Weiter stellen höhere Programmiersprachen eine Abstraktion dar, die viele Details der Ausführung verlässlich behandelt, ohne dass diese während der Programmierung beachtet werden müssen. So werden zum Beispiel die Details, wie Variablen bei Berechnungen oder dem Aufruf von Unterprogrammen geladen und gespeichert werden müssen, durch Compiler oder Interpreter korrekt behandelt, ohne dass jeder Lade- und Speichervorgang kleinteilig in der höheren Sprache beschrieben wird. Neben dieser direkt verfügbaren Abstraktion lassen die meisten höheren Sprachen zu, weitere Abstraktionen im Programm selbst vorzunehmen. Diese weiteren Abstraktionen können je nach Programmierparadigma durch Typdefinitionen, Methoden, Funktionen, Klassendefinitionen, ganze Bibliotheken usw. vorgenommen werden. Erst durch solche Abstraktionen werden Programme, die in Maschinensprache aus Millionen einzelner Instruktionen bestehen, in einer für Menschen lesbaren Beschreibungsebene erfasst. Für die Gegenstandskonstitution bedeutet dies, dass die Programme in höheren Sprachen gerade in ihrer durch Menschen lesbaren Form konstituiert werden – anstelle einer Konstitution als Reihe von direkt vom Prozessor ausführbaren Befehlen. Die

Gegenstandskonstitution ist also durch die Sprache, durch ihre syntaktischen Regeln und ihre Abstraktion von tatsächlich ausgeführten Rechenschritten geformt.

Dem Compiler (oder Interpreter) kommt dabei die wichtige Aufgabe zu, den menschenlesbaren Quellcode in einer höheren Programmiersprache zu übersetzen. Compiler sind zunächst alle Programme, die von einer Programmiersprache in eine andere übersetzen. Der hier aufgegriffene idealtypische Fall stellt dabei die Übersetzung in eine vom Computer ausführbare Sprache dar. Das kann Maschinensprache sein, also eine Reihe von Instruktionen, die direkt von einem Prozessor ausgeführt werden können. Es kann sich auch um eine maschinenunabhängige Zwischendarstellung handeln, die mit Hilfe weiterer Programme ausgeführt werden kann. Zum Beispiel werden Java-Programme vom Compiler in eine Zwischendarstellung in *Java Bytecode* übersetzt. Diese ist maschinenunabhängig und nicht vom Befehlssatz eines Prozessors abhängig. Durch maschinenabhängig implementierte *virtuelle Maschinen* kann der Bytecode auf Computern ausgeführt werden. Die virtuelle Maschine überträgt dabei den Bytecode in jeweils auf dem Prozessor ausführbare Instruktionen, bildet die Schnittstelle zum Betriebssystem, behandelt die programminterne Speicherverwaltung usw. Durch die Verwendung von virtuellen Maschinen sind bei Java die Sprachfunktionen der Maschinenunabhängigkeit und der Abstraktion teilweise voneinander getrennt: Die virtuelle Maschine stellt die Maschinenunabhängigkeit sicher und abstrahiert von den Details der Prozessorbefehle und der internen Speicherverwaltung. Sie stellt für Bytecode-Programme eine grundlegende Funktionalität bereit, die im Idealfall auf jeder Hardware, jedem Betriebssystem gleich ist. Der Compiler abstrahiert von der relativ einfachen Bytecode-Syntax und ermöglicht die Programmierung in den umfangreichen Elementen der Sprache Java. Dadurch kann die Funktion eines Java-Programms in der Terminologie der virtuellen Maschine ausgedrückt werden. Java-Programme können durch das Zusammenwirken von Compiler und virtueller Maschine als abstrakt und maschinenunabhängig konstituiert werden – insofern für die jeweils intendierte Hardware und das Betriebssystem eine funktionsfähige virtuelle Maschine verfügbar ist. Eine solche Gegenstandskonstitution ist also abhängig von stabiler, das heißt gewarteter und ständig angepasster, weiterer Software.

Das gleiche gilt für Compiler, die direkt in Maschinensprache übersetzen. Auch diese sind Gegenstand erheblicher Arbeit, um Entwicklungen sowohl der Hardware und anderer Software als auch der Sprache selbst zu berücksichtigen. Üblicherweise wird bei komplexen Compilern zwischen *Frontend*, *Middleend* und *Backend* unterschieden. Das Frontend übernimmt dabei die oben beschriebene lexikalische, syntaktische und semantische Analyse, um das Programm in eine interne Zwischendarstellung zu übertragen. Eine häufig verwendete Form der Zwischendarstellung ist ein abstrakter Syntaxbaum (*abstract syntax tree,* kurz AST). Das Programm wird dabei anhand der formalen Grammatik in eine Baumstruktur übertragen, in der jedes Programmelement als Knoten repräsentiert ist, dessen Kindknoten die Teilelemente darstellen. Das Backend produziert einen von der Zielarchitektur verarbeitbaren Code, beispielsweise in Maschinensprache oder in Java-Bytecode. Dabei wird für die Übersetzung in Maschinencode unter anderem festgelegt, welche prozessorinternen Speicher, *Register*, an welchen Stellen genau verwendet werden. Weiter werden von der Zielarchitektur abhängige Optimierungen durchgeführt, beispielsweise wird die Reihenfolge der Ausführung von Instruktionen so angepasst, dass Rechenkapazität gut ausgenutzt wird und möglichst wenige zeitintensive Speicher- und Ladeinstruktionen notwendig sind.

Das Middleend führt, ausgehend vom AST (oder einer anderen Zwischendarstellung), Optimierungen durch. Dabei ist es nicht unüblich, dass die Form der Darstellung innerhalb eines Übersetzungsprozesses mehrfach geändert wird, so dass die jeweiligen Optimierungen

durchgeführt werden können. Solche Optimierungen können zum Beispiel ungenutzte Teile des Codes entfernen, bei imperativen Sprachen beispielsweise Instruktionen, die durch den Kontrollfluss nie erreicht werden können. Durch Analyse der Zwischendarstellungen lassen sich die Anzahl benötigter Variablen und damit der benötigte Speicher bei Ausführung und tatsächlich ausgeführte Speicher- und Ladevorgänge reduzieren. Ziel der zahlreichen Optimierungstechniken sind kleinere und schnellere übersetzte Programme. Für diese Arbeit ist insbesondere relevant, welche Veränderungen sich durch die Optimierungen am Gegenstand ergeben und wie dies auf die Wahrnehmung des Programms zurückwirkt. Einerseits bezwecken die Optimierungen Veränderungen: Das in Maschinensprache, Bytecode oder eine andere Programmiersprache übersetzte Programm wird kleiner. Bei der Ausführung benötigt es eine kürzere Zeit und weniger Ressourcen (insbesondere Hauptspeicher). Andererseits wird erwartet, dass die Funktionalität des Programms sich in wesentlichen Punkten nicht ändert: Die gleiche Eingabe soll auch nach Optimierung zur gleichen Ausgabe führen, lediglich Unterschiede im zeitlichen Verlauf, also eine schnellere Ausführung, sind akzeptiert. Dem optimierten Programm wird in zentralen Aspekten die gleiche Bedeutung zugeschrieben. Inwiefern Optimierungen die Semantik verändern wird im nächsten Unterkapitel ausführlicher behandelt.

Durch die Anwendung optimierender Compiler kann auf viele manuelle Optimierungen beim Schreiben des Quellcodes verzichtet werden. Dadurch tritt die Funktion des Quellcodes, das Programm in menschenlesbarer Form darzustellen, noch stärker in den Vordergrund: Es kann ineffizient – aber dafür verständlich – formuliert werden, solange der Compiler verlässlich Ineffizienzen durch automatische Optimierung entfernt. Dadurch verändern die Optimierungen nicht nur das Verhalten des Programms in der späteren Ausführung, sie verändern auch die Art und Weise, wie Programme geschrieben werden können. Dem Quellcode kommt sowohl durch die Abstraktion höherer Programmiersprachen als auch durch die Möglichkeit der Optimierung eine signifikant andere Bedeutung zu als der maschinennahen Programmierung ohne diese Hilfsmittel: Er stellt dadurch viel stärker einen menschenlesbaren Formalismus von dem, was passieren soll, dar, als einen maschinenlesbaren Formalismus von dem, was tatsächlich bei der Ausführung passiert.

Neben den Compilern und Interpretern, die mittlerweile in praktisch jeder Softwareentwicklung eingesetzt werden (direkte Entwicklung von Maschinensprache stellt eine seltene Ausnahme dar), ist die Zahl weiterer regelmäßig verwendeter Werkzeuge und Hilfsmittel in den letzten Jahrzehnten stark gestiegen. Bereits einfache Textverarbeitungsprogramme, mit denen Programme direkt am Computer geschrieben und anschließend weiterverarbeitet werden können, stellen eine große Erleichterung gegenüber dem Stanzen von Lochkarten dar. Die heute verwendeten integrierten Entwicklungsumgebungen bieten zahlreiche weitere Werkzeuge zusätzlich zu einer speziell auf das Programmieren zugeschnittenen Textverarbeitung. Die zugeschnitte Textverarbeitung hebt zum Beispiel Wörter je nach Bedeutung in der verwendeten Programmiersprache farblich hervor, bietet eine automatische Wortvervollständigung und verwaltet auf mehrere Dateien verteilten Quelltext.

Weitere integrierte Hilfsmittel können Buildsysteme sein, die die Übersetzung und Verlinkung steuern, Werkzeuge zur Fehlersuche und -korrektur, Versionsverwaltungssysteme, Testumgebungen, Werkzeuge zur Entwicklung von Benutzeroberflächen, Anbindungen an Projekt- und Aufgabenverwaltung, Ticketsysteme zur Verwaltung von Fehlermeldungen und Änderungswünschen usw. Diese Hilfsmittel erleichtern den Umgang mit Quellcode und die Arbeit an Programmen auf vielfältige Art und Weise. Sowohl einzeln betrachtet als auch in ihrer Gesamtheit haben sie auch einen erheblichen Einfluss darauf, wie Programme als

Gegenstand wahrgenommen werden können. Beispielhaft kann dies anhand der Automatisierung von Tests und der Integration dieser in den Entwicklungsprozess illustriert werden:

Die Funktionalität von Programmen zu überprüfen und mit einem gewünschten Ergebnis zu vergleichen ist aufwändig. Der Quellcode muss dazu übersetzt und mit verschiedenen Eingaben ausgeführt werden, die Ausgaben jeweils mit dem gewünschten Ergebnis verglichen werden. Um ein hohes Vertrauen in die Korrektheit des Programms zu erreichen müssen sehr viele Tests mit möglichst heterogenen Eingaben und Parametern durchgeführt werden. Die Tests sollten dabei sowohl einzelne Teile des Programms unabhängig voneinander betreffen (*Unit Tests*), das Zusammenwirken der einzelnen Teile überprüfen (*Integration Tests*), sowie die Funktionalität des gesamten Programms sicherstellen (*System Tests*). Komplexe Programme können dabei die unangenehme Eigenschaft haben, dass Änderungen an einer Stelle des Quellcodes Auswirkungen haben, die im Zusammenwirken mit ganz anderen Teilen schließlich zu Fehlern führen. Aus diesem Grund ist es wünschenswert, dass bei jeder Änderung alle Teile des Programms sowie die Integration und das Gesamtsystem erneut überprüft werden. Solange Tests manuell durchgeführt werden ist der Aufwand in vielen Fällen kaum tragbar.

Das Aufkommen automatisierter Werkzeuge zur Durchführung sowohl der Unit Tests als auch der Integration und System Tests hat den Entwicklungsprozess maßgeblich verändert. Durch diese Werkzeuge können nun nach jeder Änderung am Quellcode alle definierten Tests automatisiert erneut durchgeführt werden. Weiter hat der Umfang an durchgeführten Tests drastisch zugenommen: Selbst kleinste Module in großer Anzahl können ohne allzu großen Aufwand regelmäßig getestet werden. Die Entwicklung von Tests ist viel enger mit der Softwareentwicklung verknüpft und kann sogar als Teil derselben gesehen werden. Die automatisierten Tests können ähnlich dem eigentlichen Quellcode formalisiert ausgedrückt werden, teilweise auch in derselben Programmiersprache. Dadurch kann mit automatisierten Tests die automatisierte Überprüfung der Funktionalität als Teil des syntaktischen Gegenstands konstituiert werden. Die Techniken zur Bearbeitung des Gegenstands haben dabei die Bedeutung dessen erweitert, was als zum Gegenstand dazu gehörend konstituiert wird. *Test Driven Development*, eine Technik zur Entwicklung von Programmen, geht noch einen Schritt weiter: Hier werden die Anforderungen an die Funktionalität bereits vor dem Schreiben des Programms als automatisierte Tests formuliert. Also wird eine formalisierte normative Erwartung an den Entwicklungsprozess sowohl als Grundlage zur Herstellung des Gegenstands Programm als auch als Indikator für dessen fehlerfreies Funktionieren herangezogen. Die Tests werden dabei als Übersetzung oder Ableitung aus der Spezifikation der gewünschten Funktionalität betrachtet.

Eine weitere Technik, die sowohl integriert in eine Entwicklungsumgebung als auch separat dazu angewendet wird, sind Versionsverwaltungssysteme. Wie im letzten Unterkapitel beschrieben drücken Versionen und Versionsnummern gleichzeitig einen Zusammenhang und ein Unterscheidungsmerkmal von Programmen aus. Einerseits werden verschiedene Versionen eines Programms als derselbe Gegenstand konstituiert: Das Programm hat den gleichen Namen, es erfüllt zumindest teilweise die gleiche, vergleichbare oder ähnliche Funktionalität. Eine neuere Version eines Textverarbeitungsprogramm ist meist auch ein Textverarbeitungsprogramm. Oft wird eine Rückwärtskompatibilität erwartet: Z.B. sollen Dateien, die als Ausgabe einer älteren Programmversion erstellt wurden auch mit der neueren Version lesbar sein. Gleichzeitig können die Versionen auch als unterschiedliche Gegenstände konstituiert werden: Die Funktionalität wurde erweitert, Fehler wurden behoben, das Programm kann mit weiteren Programmen verknüpft zusammenarbeiten usw.

Syntaktisch sind unterschiedliche Versionen eines Programms (fast) immer unterschiedlich: Der Quellcode wurde geändert. Um solche Änderungen zu koordinieren und nachvollziehen zu können werden Versionsverwaltungssysteme in den Entwicklungsprozess eingebunden. Diese Systeme speichern üblicherweise jede Version des Quellcodes und machen sie jederzeit für die Beteiligten verfügbar, protokollieren Änderungen und dienen als Koordinationswerkzeug für alle, die am jeweiligen Programm arbeiten. Gegenwärtig verwendete Versionsverwaltungssysteme wie *Git* oder *Mercurial* lassen zum Zweck dieser Koordination Verzweigungen in der Entwicklung zu, zum Beispiel einen neuen Strang von Versionen, in dem eine bestimmte Funktionalität entwickelt wird. Solche Verzweigungen können wieder zusammengeführt werden und sind oftmals fester Bestandteil des Softwareentwicklungsprozesses. Durch die Verwendung solcher Systeme kann ein Computerprogramm anstelle einer einzelnen Version auch als Repositorium einer Menge von Versionen konstituiert werden. In diesem Fall sind alle Änderungen, die über die Zeit hinweg stattgefunden haben, nachvollziehbarer Teil des Gegenstands. Die verschiedenen Versionen und Abhängigkeiten derselben voneinander lassen sich dabei als Graph darstellen, der die jeweils zur Herstellung einer Version verwendeten Versionen als durch Kanten verbundene Knoten enthält. Solche Graphen werden als Versionsgraph (manchmal auch Versionsbaum genannt, der Graph muss dabei jedoch mathematisch kein Baum sein) bezeichnet und visualisieren die Historie von Änderungen über die Versionen hinweg. Die „Geschichte“ eines syntaktischen Programms kann also als Teil des Gegenstands wahrgenommen und auch technisch visualisiert werden.

Im Abschnitt zu Tests wurde kurz angeführt, dass die Spezifikation den Charakter einer normativen Erwartung an den Quellcode hat: Sie drückt aus, welche Funktionalität das Programm haben soll, wie es auf bestimmte Eingaben reagieren soll, welche Anforderungen an das Verhalten des Computers bei der Ausführung des Programms gestellt werden. Die Spezifikation legt eine Norm fest, nach der der Quellcode bewertet wird. Er *funktioniert*, oder er ist *fehlerhaft*. Diese Norm kann formalisiert werden, etwa in Form von Tests oder einer formalen Beschreibung der Funktionalität. Während sich ethische und rechtliche Normen direkt auf die Funktionalität eines Programms beziehen (z.B. Datenschutzbestimmungen) geht es hier um das Verhältnis vom syntaktischen zum semantischen Gegenstand: Tut das Programm was es tun soll? Diese Frage wird im Kapitel zum Programm als Assoziation aufgegriffen.

Direkt auf den Quellcode als Gegenstand beziehen sich einige ästhetische Normen. So gibt es Konventionen, Richtlinien und auch Diskussionen dazu, wie Quellcode geschrieben sein sollte, damit er gut lesbar, wartbar und erweiterbar ist. Ein Programm kann als elegant, als gut geschrieben, sauber konstituiert werden, oder aber der Code erscheint als Durcheinander, Murks, Frickelei, *quick and dirty*. Für jeweilige Sprachparadigmen und einzelne Sprachen existieren Zusammenfassungen einiger dieser Normen, für Java zum Beispiel die 1997 von Sun Microsystems veröffentlichten Coding Conventions[87] und die 2013 von Fred Long et al. veröffentlichten Coding Guidelines[88]. Eine frühe Auseinandersetzung mit dem Verhältnis von Stil zu Lesbarkeit findet sich in *The Elements of Programming Style*[89]. Neben dem Befolgen von Regeln, um die Lesbarkeit und Wartbarkeit von Programmen sicherzustellen, umfasst die Frage nach ästhetischen Urteilen auch, wie ein auf Programme bezogener Begriff von Schönheit aussehen könnte. Die Thematik kann hier nur kurz

87 Diese wurden auf Oracles Website zur Verfügung gestellt. Da sie seit 1999 nicht an Änderungen der Sprache angepasst wurden gelten sie als überholt. Zur Zeit abrufbar unter http://www.oracle.com/technetwork/java/codeconventions-150003.pdf (abgerufen am 05.08.2018).

88 Long, Fred, et al.: Java coding guidelines: 75 recommendations for reliable and secure programs. Addison-Wesley. 2013.

89 Kernighan, Brian W., and Phillip James Plauger: The elements of programming style. 2nd Ed. McGraw-Hill. 1978.

angerissen werden, bietet sich jedoch für nachfolgende Untersuchungen an. Durch den hier vorgeschlagenen Programmbegriff kann dabei unterschieden werden, inwiefern sich Urteile über Schönheit auf den syntaktischen, den semantischen oder den eingebetteten Gegenstand beziehen: Eine besonders schöne Benutzeroberfläche muss nicht zwangsweise auch in sauberem Code implementiert werden, wenn ein Algorithmus als elegant beurteilt wird sagt dies noch nichts darüber aus, ob der Code auch lesbar geschrieben ist. Donald Knuth beschreibt in der Einleitung der Buchreihe *The Art of Computer Programming* das Programmieren als ästhetische Erfahrung, vergleichbar mit der Komposition von Poesie oder Musik[90]. In diesem Fall wird besonders deutlich, dass sich Kunst- und Schönheitsbegriff dabei auf die Semantik beziehen sollen, da Knuth in der Buchreihe die in der Programmierung verwendeten Algorithmen vorstellt. Viele Perspektiven auf die Frage nach Schönheit im Code finden sich in *Beautiful Code*[91], hier werden Beispiele von als besonders schön empfundenem Code von Softwareentwickler_innen vorgestellt. In der Einleitung beschreibt Greg Wilson:

> „[…] I saw that programs could be more than just instructions for computers. They could be as elegant as well-made kitchen cabinets, as graceful as a suspension bridge, or as eloquent as one of George Orwell's essays."[92]

Bis zu dieser Stelle wurde unter einem syntaktischen Programm Quellcode verstanden, in dem eine bestimmte Funktionalität oder ein bestimmtes Verhalten bei der Ausführung ausgedrückt wird. In der Computertechnik gibt es mindestens zwei Entwicklungen, die diese Abgrenzung infrage stellen. Erstens verschwimmt durch die Verfügbarkeit programmierbarer Hardware und ausgereifter Hardwarebeschreibungssprachen die Grenze zwischen Hardware und Software zunehmend. Die Hardwareentwicklung greift durch das Vorliegen der Hardwarebeschreibung in einer formalen Sprache (z.B. Verilog) auf Techniken zurück, die große Parallelen zur Softwareentwicklung aufweisen. Integrierte Entwicklungsumgebungen, automatisierte Tests, Versionsverwaltungssysteme, formale Verifikation usw. werden auch in der Hardwareentwicklung eingesetzt, auch der Hardwareentwurf liegt als sprachlicher Gegenstand vor. Zweitens liegt der Quellcode von Programmen nicht immer rein sprachlich vor. Die im Compiler verwendete Zwischendarstellung eines abstrakten Syntaxbaums kann beispielsweise auch als Quellcode begriffen werden, neben den sprachlichen Elementen ist hier deren Verortung in der mathematischen Baumstruktur von Bedeutung. Eine andere Darstellungsweise, bei der die Bedeutung durch eine Kombination der Verortung in einem Graphen und sprachliche Elemente ausgedrückt wird, sind Kontrollflussdiagramme. Hier wird die Ausführungsreihenfolge von Codeblöcken graphisch dargestellt. Visuelle Programmierung greift auf graphische Elemente bei der Entwicklung von Programmen zurück. Die Graphen lassen sich dabei zwar sprachlich ausdrücken (letztendlich immer als Folge von Nullen und Einsen in Maschinensprache), bei der Arbeit an den Programmen werden sie jedoch nicht rein sprachlich konstituiert.

Im hier vorgeschlagenen Programmbegriff zählen beide Grenzfälle auch als syntaktisches Programm. Hardwarebeschreibungen werden in der Wahrnehmung mit vielen Parallelen zum Quellcode in einer Programmiersprache konstituiert. Auch die Arbeit an diesen Beschreibungen als Tätigkeit ist bereits durch die Begrifflichkeit *programmierbarer* Hardware als Programmieren gefasst. Die Entwicklungsmethoden und -werkzeuge weisen viele Parallelen auf und überschneiden sich teilweise. Insbesondere lässt sich jede Hardwarebeschreibung auch als Programm auffassen: Durch Simulation der Hardware kann

90 Knuth, Donald: The Art of Computer Programming. Vol. 1 / Fundamental Algorithms. Addison-Wesley. 1968.

91 Oram, Andrew, and Greg Wilson: Beautiful code. O'Reilly, 2007.

92 Ebda. S. XV.

das durch die Beschreibung vorliegende syntaktische Programm ausgeführt werden, die modellierten Berechnungen werden durch den Simulator durchgeführt. Programmiersprachen und Hardwarebeschreibungssprachen können durch Compiler in vielen Fällen ineinander überführt werden. Die grenzen zwischen den Sprachen sind fließend, beispielsweise gibt es mit SystemC eine Programmbibliothek für die Programmiersprache C++, die viele Elemente der Hardwaremodellierung implementiert. Die Zwecke, zu denen Hardwarebeschreibungssprachen und Programmiersprachen verwendet werden, unterscheiden sich zwar – und weder sind Programmiersprachen ohne größere Erweiterungen für die Hardwareentwicklung geeignet noch Hardwarebeschreibungssprachen zum Schreiben komplexer Programme – allerdings überwiegen die Parallelen. Beides sind formale Sprachen, beide haben eine durch Berechnung gekennzeichnete Semantik, beide werden in ähnlichen Entwicklungsprozessen verwendet und die Produkte sind nur in komplexen technischen Rahmenbedingungen unter zahlreichen Voraussetzungen anwendbar.

Auch visuelle Programme, Kontrollflussgraphen usw. werden hier als syntaktisches Programm aufgefasst[93]. Solche Beschreibungen von Programmen greifen erstens oft auf sprachliche Elemente zurück, die wiederum den gleichen grammatischen Regeln unterliegen wie Programme nichtgrafischer Programmiersprachen. Zweitens unterliegt auch die Darstellung, etwa als Graph, Regeln, welche die Gültigkeit ausgedrückter Programme festlegen. Eine solche Regel für Kontrollflussgraphen wäre beispielsweise, dass Kanten gerichtet sind und immer genau zwei Knoten (also Codeblöcke) miteinander verbinden. Drittens schließlich kommt den Darstellungen, die auch auf eine bildliche Anordnung zurückgreifen, die gleiche Funktion zu wie dem Quellcode als einer rein sprachlichen Darstellung. Die Regeln, nach denen visuelle Programme konstituiert werden, können in diesem Sinne als Syntax aufgefasst werden, die konstituierten Gegenstände sind syntaktische Programme.

93 Vgl. z.B. Harman, Mark: Why Source Code Analysis and Manipulation Will Always Be Important. In 10th IEEE Working Conference on Source Code Analysis and Manipulation (SCAM). 2010. S. 7-19.

3.3 Das Programm als semantischer Gegenstand

3.3.1 Maschinelle Semantik

Wird ein Computerprogramm als syntaktischer, sprachlicher, Gegenstand konstituiert, so ist dabei eine Bedeutung der Sprache appräsentiert. Ein zentraler Teil dieser Bedeutung ist dabei, welche Berechnungen – welches Verhalten einer Maschine – im syntaktischen Gegenstand ausgedrückt ist. Diese Bedeutung ist abstrakt: Sie wird unabhängig vom physischen Gegenstand konstituiert, über den die Sprache wahrgenommen werden kann. Einer Programmdatei, die über eine Textverarbeitung am Monitor gelesen werden kann, wird die gleiche Bedeutung zugeschrieben wie einem Ausdruck derselben Datei. Gleichzeitig ist die Bedeutung unabhängig von konkreten Maschinen, weiteren Programmen und Anwender_innen, mit denen das Programm zur Laufzeit interagiert. Die Bedeutung des Quellcodes unterscheidet sich vom Programm, welches zur Laufzeit eingebettet konstituiert wird. Wird die Bedeutung als Verhalten einer Maschine verstanden, so geht es dabei um eine abstrakte Maschine: Dem Quellcode wird vermittelt über die Programmiersprache dieselbe Bedeutung zugeschrieben – unabhängig davon welcher konkrete Computer später eine Ausführung ermöglicht, unabhängig davon ob das Programm überhaupt je ausgeführt wird. In diesem abstrakten Sinne wird die Dimension der Programmbedeutung, die sich auf seine Funktion bezieht, hier als maschinelle Semantik verstanden.

Aufgrund der formalen Sprachen, in denen Computerprogramme ausgedrückt sind, lässt sich auch die maschinelle Semantik mit formalen Mitteln beschreiben. Eine Beschreibung mit den formalen Mitteln der Mathematik bringt den Vorteil mit sich, dass sich weitere mathematische Techniken auf das Programm anwenden lassen: Mit einer formalen Semantik können Beweise über Eigenschaften eines Programms geführt werden, z.B. Korrektheitsbeweise in formalen Verifikationen. Dies wird dadurch möglich, dass formale Semantiken auf die sprachlichen Mittel der Mathematik zurückgreifen. Verschiedene formale Semantiken, z.B. operationelle und denotationelle Semantiken, können danach differenziert werden, a*uf welche Art und Weise* das Programm erfasst und mit welchen Mitteln der Mathematik es beschrieben wird.

Es existiert nicht zu jeder Programmiersprache eine formale Sprachsemantik. Für Java z.B. ist die Sprachspezifikation größtenteils in natürlicher Sprache geschrieben. Zwar wurden später verschiedene formale Semantiken für Teile der Sprache vorgeschlagen[94], maßgeblich für den Compilerbau bleibt jedoch immer die von Oracle veröffentlichte natürlichsprachliche Spezifikation. Die maschinelle Semantik kann also als formal konstituiert werden, sie kann auch natürlichsprachlich konstituiert werden. Beide Vorgehensweisen führen zu einem abstrakten Gegenstand, der das Verhalten einer abstrakten Maschine beschreibt. Für einzelne Sprachen können dabei auch gleichzeitig formale und informale Beschreibungen der Bedeutung existieren. Üblicherweise wird eine der Beschreibungen als normativ bindend festgelegt: Weichen zwei Beschreibungen voneinander ab, so ist hierdurch festgelegt, dass die nicht bindende Beschreibungen fehlerhaft ist.

Dies führt zu der Frage, aus welchen Gründen die Entscheidung für eine formale

94 Z.B. Drossopoulou, Sophia, und Eisenbach, Susan: Describing the semantics of Java and proving type soundness. In: Alves-Foss, Jim: Formal Syntax and Semantics of Java. Springer, 1999. S. 41-80.

Neben diesem Artikel enthält der Sammelband weitere formale Semantiken jeweils für Teile der Sprache. Insbesondere die Nebenläufigkeit wird häufig ausgeklammert.

Semantik bzw. eine informale Semantik zur Sprachspezifikation getroffen wird. Frederick P. Brooks greift dieses Problem in *The Mythical Man-Month* auf. Dabei zielt er zwar zunächst auf die Spezifikation von Hardware ab (Brooks reflektiert insbesondere die Erfahrungen, die er bei der Entwicklung der IBM System/360 Computer machte), lässt aber die Übertragung auf Softwareprojekte explizit zu.

> „Let us examine the merits and weaknesses of formal definitions. As noted, formal definitions are precise. They tend to be complete; gaps show more conspicuously, so they are filled sooner. What they lack is comprehensibility. With English prose one can show structural principles, delineate structure in stages or levels, and give examples. One can readily mark exceptions and emphasize contrasts. Most important, one can explain *why*. The formal definitions put forward so far have inspired wonder at their elegance and confidence in their precision. But they have demanded prose explanations to make their content easy to learn and teach. For these reasons, I think we will see future specifications to consist of both a formal definition *and* a prose definition.“[95]

Neben der formalen oder informalen sprachlichen Beschreibung kann eine Sprachspezifikation auch eine Beispielimplementierung eines Compilers für die Sprache oder Testprogramme mit einer Beschreibung der erwarteten Funktionalität enthalten. Auf das Beispiel Java bezogen stellt Oracle neben der informalen Sprachspezifikation auch einen Compiler für Java als Referenzimplementierung zur Verfügung. Für diese Arbeit ist es wichtig, festzuhalten, dass die maschinelle Semantik als mathematisches Objekt, als formale Semantik, erfasst werden kann – aber auch informale Beschreibungen und andere Formen der Spezifikation möglich sind.

Formale Semantiken können abhängig davon, wie sie genau die Bedeutung mathematisch erfassen, zu Klassen oder Familien von Semantiken gruppiert werden. Eine solche Unterteilung kann beispielsweise anhand der drei größeren Familien operationelle Semantik, denotationelle Semantik und axiomatische Semantik erfolgen[96]. Operationelle Semantiken beschreiben die Ausführung des Programms auf einer abstrakten Maschine. Diese abstrakte Maschine wird dabei durch einen Zustand beschrieben, dieser Zustand enthält beispielsweise alle momentanen Variablenwerte. Das Programm stellt sich als Folge von Zustandsübergängen dar. Die mathematische Beschreibung ist also eine Abstraktion der Ausführung auf einem physischen Computer[97]. Die Beschreibung fokussiert, wie die Zustandsübergänge einer Ausführung genau ablaufen. Die maschinelle Semantik des Programms wird in dieser formalen Beschreibung als Abfolge von einzelnen Schritten, von Zustandsübergängen, konstituiert.

Denotationelle Semantiken dagegen beschreiben das Programm als Verkettung mathematischer Funktionen. Im Gegensatz zur operationellen Semantik zielt diese Beschreibung auf den Effekt, auf die Berechnung des Programms ab. Dieser Effekt wird dabei als Verknüpfung von Start- und Endzustand beschrieben[98], dies kann als abstrakte Beschreibung der Relation zwischen Ein- und Ausgabe eines Programms bei der Ausführung aufgefasst werden. Denotationelle Beschreibungen sind kompositional, das heißt dass die

95 Brooks, Frederick P.: The Mythical Man-Month. Essays on Software Engineering. Anniversary Edition. Reading, Addison-Wesley. 1995 (1975). S. 63f.

96 siehe z.B. Nielson, Hanne Riis, und Nielson, Flemming: Semantics with applications. Springer, 2007 (1992). S. 2ff.

97 Vgl. Ebda. S. 3.

98 Vgl. Ebda. S. 85.

Bedeutung eines zusammengesetzten syntaktischen Gegenstands sich aus den denotationellen Beschreibungen der einzelnen Teile des Gegenstands sowie der Art ihrer Zusammensetzung herleiten lassen. Die maschinelle Semantik wird dabei als Wirkung konstituiert, zum Beispiel als Ausgabe, die ein Programm bei einer gegebenen Eingabe produziert.

Axiomatische Semantiken beschreiben Computerprogramme in Aussagenlogik. Das Programm stellt sich hierbei als Verknüpfung der vor der Ausführung gültigen Vorbedingungen mit den nach der Ausführung gültigen Nachbedingungen dar. Diese Darstellung eignet sich insbesondere, um partielle Korrektheit von Programmen nachzuweisen[99]. Partielle Korrektheit bedeutet dabei, dass unter der Bedingung, dass das Programm terminiert, bestimmte Eigenschaften gültig sind. Diese Eigenschaften werden als Assertionen (Zuschreibungen) ausgedrückt: Gilt eine bestimmte Vorbedingung vor der Ausführung einer Programminstruktion oder eines Teilprogramms und diese Instruktion oder das Teilprogramm terminiert, so ist nach der Ausführung eine bestimmte Nachbedingung wahr. Vorbedingung und Nachbedingung sind dabei als logische Aussagen über den Zustand der abstrakten Maschine formuliert, zum Beispiel als Aussage darüber, welchen Wert eine Variable zu einem bestimmten Zeitpunkt einnimmt. Das Programm wird dabei als Folge von Übergängen bezüglich der Wahrheitswerte einer Menge von Assertionen konstituiert. Dies stellt eine starke Abstraktion dar, weil sich die Assertionen jeweils auf die Eigenschaften konzentrieren können, über die ein Korrektheitsbeweis geführt werden soll. Neben der Sprache, in der Aussagen über Programme formuliert werden, enthält eine axiomatische Semantik ein Inferenzsystem, um diese Aussagen beweisen zu können.

Da sich für ein und dieselbe Programmiersprache mehrere formale Semantiken konstruieren lassen spielt die Frage nach dem Verhältnis verschiedenartiger Semantiken zueinander eine wichtige Rolle in der theoretischen Informatik. Für eine axiomatische Semantik kann beispielsweise die Zuverlässigkeit und Vollständigkeit im Bezug auf eine operationelle oder denotationelle Semantik nachgewiesen werden[100]. Zuverlässigkeit beschreibt dabei die Eigenschaft, dass die bewiesenen Korrektheitsaussagen auch tatsächlich auf die modellierte Semantik zutreffen. Vollständigkeit beschreibt die Eigenschaft, dass alle für die modellierte Semantik zutreffende Aussagen über das Inferenzsystem bewiesen werden können. Für eine denotationelle Semantik kann nachgewiesen werden, dass sie eine operationelle Semantik adäquat abbildet, also unterschiedliche Folgen von Zustandsübergängen auch durch unterschiedliche Denotationen erfasst werden. Trifft auch die umgekehrte Relation zu, dass unterschiedliche Denotationen immer auch unterschiedliche operationelle Semantiken beschreiben, spricht man von vollständiger Abstraktion (*full abstraction*). Der Nachweis dieser Eigenschaft kann sehr aufwändig sein. Weiter ist die Eigenschaft oft dadurch nicht gegeben, dass die denotationelle Semantik zu ausdrucksstark ist und Objekte darstellen kann, denen keine ausführbare Zustandsfolge entspricht[101]. Für den Programmbegriff ist dabei relevant, dass auch eine formale Bedeutung von konkreten Programmen auf völlig unterschiedliche Arten konstituiert werden kann. Die verschiedenen Formalismen, die jeweils eine Beschreibung des Programms ermöglichen, müssen nicht äquivalent sein. Sie unterscheiden sich sogar in vielen Fällen absichtlich, etwa wenn Assertionen ein Programm auf als relevant gesetzte Eigenschaften reduzieren.

In jedem Fall wird die Funktion des Programms – hier als maschinelle Semantik beschrieben – als abstrakter Gegenstand konstituiert. Physische Gegenstände wie z.B. Papier

99 Siehe Nielson, Hanne Riis, und Nielson, Flemming: Semantics with applications. Springer, 2007 (1992). S. 169ff.

100 Siehe ebda. S. 183ff.

101 Vgl. Hyland, J. M. E. und Ong, C.-H. L: On Full Abstraction for PCF: I, II, and III. In: Information and Computation, vol. 163, no. 2. 2000. S. 292f.

oder Dateien appräsentieren die Bedeutung zwar über die auf ihnen festgehaltene Sprache, die Semantik hingegen ist kein physischer Gegenstand, der verändert werden oder zu einer bestimmten Zeit an einem bestimmten Ort festgemacht werden kann. Vielmehr verweist ein physisches Programm, an dem Änderungen vorgenommen werden, anschließend auf eine andere Semantik. Nur durch den abstrakten Charakter und die damit verbundene Permanenz können mathematische Beweise über maschinelle Semantiken in ihrer formalen Fassung vorgenommen werden.

Dieser abstrakte Charakter der Semantik beinhaltet ein philosophisches Problem, nicht nur im Bezug auf Computerprogramme. Dieses Problem kann an dieser Stelle nur kurz angerissen werden, allerdings zeigt sich im Bezug auf Computerprogramme durch den phänomenologischen Zugang dieser Arbeit eine neue Perspektive auf das Problem. Deshalb soll, ohne die Fragen abschließend zu behandeln, kurz aufgezeigt werden, welchen Beitrag der hier vorgeschlagene Begriff leisten könnte. Paul Benacerraf verdeutlicht das Problem an zwei schwierig zu vereinbarenden Anliegen:

> „It is my contention that two quite distinct kinds of concerns have separately motivated accounts of the nature of mathematical truth: (1) the concern for having a homogeneous semantical theory in which semantics for the propositions of mathematics parallel the semantics for the rest of the language, [...] and (2) the concern that the account of mathematical truth mesh with a reasonable epistemology. It will be my general thesis that almost all accounts of the concept of mathematical truth can be identified with serving one or another of these masters *at the expense of the other*."[102]

Im Zentrum dieses Problems steht die Frage, ob mathematische Sprache überhaupt eine Semantik hat – also ob die Formalismen semantisch auf abstrakte mathematische Objekte verweisen. Wird die Frage verneint, so ist Mathematik lediglich syntaktische Manipulation. Die Zeichen dieser Sprache verweisen auf nichts, mathematischen Sätzen kommt im Gegensatz zu Sätzen natürlicher Sprache keine Bedeutung zu. Wahrheit ist in diesem Fall nicht definiert, es gibt lediglich die syntaktische Beweisbarkeit mathematischer Sätze: Gültig ist, was syntaktisch herleitbar ist, also mittels syntaktischer Manipulation aus Axiomen gefolgert werden kann. Diese Lösung ist insofern nicht zufriedenstellend, dass ein unüberwindbarer Bruch zwischen mathematischer und natürlicher Sprache daraus folgt – und dass intuitionistische Vorstellungen in der auf Syntax reduzierten Mathematik ausgeblendet werden und unerklärbar bleibt, warum die Mathematik überhaupt anwendbar ist, sie z.B. in den Naturwissenschaften mit großem Erfolg auf die Welt bezogen werden kann. Benacerraf verdeutlicht den Bruch zwischen natürlicher und mathematischer Sprache an zwei Sätzen, die sich bei der Ablehnung mathematischer Semantik grundlegend unterscheiden und eine völlig unterschiedliche Struktur von Gültigkeitsbedingungen voraussetzen:

> „(1) There are at least three large cities older than New York.
>
> (2) There are at least three perfect numbers greater than 17."[103]

Ohne mathematische Semantik ist der erste Satz aufgrund seines Bezugs zu empirischen Gegenständen wahr, der zweite aufgrund syntaktischer Herleitbarkeit. Wird hingegen eine mathematische Semantik angenommen, so führt dies zu einer Art platonischem Realismus: Die Syntax verweist auf mathematischen Objekte, die ihrerseits weder räumlich

102 Benacerraf, Paul: Mathematical Truth. In: The Journal of Philosophy, Vol. 70, No. 19. 1973. S. 661.

103 Ebda. S. 663.

noch zeitlich gegeben, also abstrakt sind. Diese platonische Antwort lässt zwar eine zufriedenstellende Homogenität der Bedeutung von Sprache zu, führt jedoch zu einem weiteren Problem, das später als *Benacerraf-Field Challenge* bezeichnet wurde. Diese epistemologische Herausforderung drückt sich in der Frage aus, wie überhaupt gesichertes Wissen über mathematische Objekte erlangt werden kann, wenn diese abstrakt sind. Benacerraf formuliert dies explizit mithilfe einer kausalen Theorie des Wissens[104], das Problem bleibt jedoch auch bei Ablehnung dieser Theorie bestehen.

Raymond Turner bezieht im Aufsatz *Understanding Programming Languages* diese Herausforderung direkt auf die Semantik von Computerprogrammen[105]. Er unterscheidet dabei zwischen operationeller und denotationeller Semantik: Operationelle Semantik übersetzt die Programmiersprache in eine andere Sprache (häufig auf dem Lambda-Kalkül basierend) und legt darüber eine Bedeutung der Programme fest. Denotationelle Semantik dagegen übersetzt direkt in die mathematische Mengenlehre, verweist also auf abstrakte Objekte. Diese Unterscheidung, die letztendlich durch die Modellierung des Lambda-Kalküls in der Mengenlehre aufgelöst wird, spielt für den hier vorgeschlagenen Programmbegriff nur eine untergeordnete Rolle: Die Semantiken werden als abstrakt verstanden unabhängig davon, in welchem Formalismus sie ausgedrückt sind bzw. ob sie überhaupt formal gegeben oder informal durch natürliche Sprache spezifiziert werden. Das Problem erstreckt sich also auf jede Form maschineller Semantik.

Turner betrachtet dieses Problem in seinem Zusammenhang mit Ludwig Wittgensteins Regelfolgeproblem[106]. Während Benacerraf die Möglichkeit problematisiert, Wissen über die abstrakten Objekte der Mathematik zu erlangen, zeigt Wittgenstein die Problematik auf, dass ein abstraktes Objekt (eine Regel) nicht kausal menschliches Handeln bestimmen kann – jedes Handeln wäre mit einer gegebenen Regel in Einklang zu bringen[107]. Turner verwendet dieses Argument gegen den auf die Semantik von Computerprogrammen bezogenen Platonismus.

> „This throws some light on the semantic issues surrounding programming languages. Any formally adequate mathematical theory can serve as the semantic vehicle for defining programming languages. What programming languages are taken to mean will be informed, indeed constituted, by the *form of life* that is the practice of the of the chosen mathematical theory - be it the Lambda Calculus, set-theory or something else.“[108]

Das Problem des Regelfolgens gibt es für alle hier vorgestellten maschinellen Semantiken. Formale und informale sprachliche Beschreibungen der Semantik bedürfen einer Interpretation, um sie anwendbar zu machen. Ist eine Sprachspezifikation durch einen konkreten Compiler gegeben, löst dies das Problem auch nicht: Saul A. Kripke zeigt in seiner Interpretation des Regelfolgeproblems, dass auch ein Programm als Definition einer Regel das Paradoxon nicht auflösen kann[109].

Turners Behandlung des Problems über Wittgensteins Begriff der Lebensform verortet

104 Benacerraf, Paul: Mathematical Truth. In: The Journal of Philosophy, Vol. 70, No. 19. 1973. S. 671.

105 Turner, Raymond: Understanding programming languages. In: Minds and Machines Vol. 17 Iss. 2. 2007. S. 203-216.

106 Ebda. S. 210f.

107 Wittgenstein, Ludwig: Philosophische Untersuchungen. 6. Auflage. Suhrkamp , 2013 (2003). Erstausgabe 1953. S. 133f.

108 Turner, Raymond: Understanding programming languages. In: Minds and Machines Vol. 17 Iss. 2. 2007. S. 214.

109 Kripke, Saul A.: Wittgenstein über Regeln und Privatsprache. Suhrkamp, 1987. Englische Erstausgabe 1982. S. 47ff.

den Status der Semantik von Computerprogrammen in der Praxis menschlicher Handlungen. Das Lambda-Kalkül beispielsweise wirkt nicht als abstrakter Gegenstand, der auf ungeklärte Art und Weise den Compilerbau beeinflusst und darüber schließlich Wirkung im konkreten Verhalten physischer Maschinen zeigt. Vielmehr ist das Lambda-Kalkül eine stetig reproduzierte Art und Weise, wie Bedeutung von Computerprogrammen aufgefasst werden kann. Die Übereinstimmung in der Sprache, in ihrer Bedeutung, kann dabei als gemeinsame Lebensform verstanden werden:

> „241. »So sagst du also, daß die Übereinstimmung der Menschen entscheide, was richtig und was falsch ist?« - Richtig und falsch ist, was Menschen *sagen*; und in der *Sprache* stimmen die Menschen überein. Dies ist keine Übereinstimmung der Meinungen, sondern der Lebensform.“[110]

Der hier vorgeschlagene Programmbegriff fasst die Semantik zwar als abstrakt auf, jedoch nicht in dem platonischen, ontologischen Sinne, den Benecerraf und Turner kritisieren. Als abstrakt werden im hier gewählten phänomenologischen Zugang diejenigen Gegenstände verstanden, die in der Konstitution als abstrakt intendiert werden. Auch solche Gegenstände werden von Menschen konstituiert, dadurch ist auch ein menschlicher Zugang zu den abstrakten Objekten gegeben. Die Konstitution eines Gegenstands als abstrakt schreibt diesem keine physischen Eigenschaften zu. Die Programmsemantik kann nicht zu einer bestimmten Zeit an einem bestimmten Ort festgemacht werden. Auch eine physische Kausalität, z.B. in dem Sinne dass die Semantik direkt das Maschinenverhalten determiniert, wird dem Gegenstand nicht zugeschrieben.

Benacerrafs epistemologische Herausforderung stellt sich durch diese Fassung des abstrakten Charakters mathematischer (bzw. semantischer) Objekte anders dar: Über die mathematischen Objekte kann zwar Wissen erlangt werden, da die Konstitution menschlicher Erfahrung einen direkten Zugang bietet. Allerdings kommt ihnen dadurch auch nur über die sie konstituierenden Menschen Kausalität zu: Ein syntaktisches Programm mit einer bestimmten Semantik verhält sich in der Ausführung nicht allein dadurch entsprechend dieser Semantik, dass es in einer bestimmten Sprache geschrieben wurde. Das Verhalten kann erst dadurch Schritte vollziehen, die an der gleichen Semantik orientiert sind, dass der Compilerbau sich an der gleichen Sprachsemantik orientiert, und dass die physische ausführende Maschine so konstruiert ist, dass sie sich entsprechend der abstrakten Maschine der Semantik verhält. Die entscheidende Frage ist in dieser Fassung nicht, wie wir Wissen über einen abstrakten Gegenstand erlangen können, sondern vielmehr, wie wir dazu kommen, genau den gleichen abstrakten Gegenstand zu konstituieren.

Die Verknüpfung von Benacerrafs Herausforderung und Wittgensteins Regelfolgeproblem ist also hier deutlich enger. Und das Regelfolgeproblem ist dadurch für Computerprogramme in besonderer Weise brisant, dass das Funktionieren komplexer computergesteuerter Systeme nur durch ein gleichförmiges auf die Semantik bezogenes Regelfolgen einer Vielzahl von Menschen möglich ist. Vom Design der IDE über den Compilerbau, die Spezifikationen anderer Programme in der Laufzeitumgebung und Schnittstellen (insbesondere zum Betriebssystem) bis zur Konstruktion der physischen Maschinen müssen eine Vielzahl unterschiedlicher Semantiken in der gleichen Weise menschliches Handeln strukturieren, damit im Ergebnis funktionierende Anwendungen ausgeführt werden können. Diese Übereinstimmung in der Lebensform bleibt dabei erklärungsbedürftig. Einen kleinen Teil dieser Erklärung kann das Unterkapitel zum Programm als Assoziation beisteuern, dort wird die gesellschaftliche Bedingtheit der

110 Wittgenstein, Ludwig: Philosophische Untersuchungen. 6. Auflage. 2013 (2003). Erstausgabe 1953. Suhrkamp. S. 145.

gleichförmigen Orientierung an einer Semantik aufgegriffen.

Die maschinelle Semantik wird als abstrakt intendiert, unabhängig davon, ob diese nun formal oder informal gegeben ist. Ein wichtiger Unterschied ist hierbei, dass formale Semantiken mathematische Schlussfolgerungen zulassen. Informale Semantik gibt zwar auch Aufschluss darüber, was ein Programm auf einer (von konkreten physischen Verwirklichungen abstrahierten) Maschine tut, lässt jedoch keine mathematische Beweisführung zu. Aus welchem Grund wird häufig auf informale Fassungen der Programmbedeutung zurückgegriffen? Kenneth Slonneger und Barry L. Kurtz beschreiben in einer Einführung zu Action Semantics, einer auf pragmatische Anwendbarkeit abzielenden formalen Semantik, die Schwierigkeiten mit vielen formalen Herangehensweisen:

> „In spite of the arguments for relying on formal specifications of programming languages, programmers generally avoid them when learning, trying to understand, or even implementing a programming language. They find formal definitions notationally dense, cryptic, and unlike the way they view the behavior of programming languages. Furthermore, formal specifications are difficult to create accurately, to modify, and to extend. Formal definitions of large programming languages are overwhelming to both the language designer and the language user, and therefore remain mostly unread.
>
> Programmers understand programming languages in terms of basic concepts such as control flow, bindings, modifications of storage, and parameter passing. Formal specifications often obscure these notions to the point that the reader must invest considerable time to determine whether a language follows static or dynamic scoping and how parameters are actually passed. Sometimes the most fundamental concepts of the programming language are the hardest to understand in a formal definition.“[111]

Diese Kritik ist dahingehend aufschlussreich, dass das Verstehen und Beherrschen einer Programmiersprache offensichtlich nicht alleine als Kenntnis ihrer Semantik erfasst werden kann. Vielmehr wird ein Wissen über *basic concepts* aufgeführt, das in einer formalen Behandlung der Programmbedeutung nicht leichter, sondern schwieriger zugänglich sein kann. Das Vorgehen, Programmiersprachen über ihre Konzepte und das Verhältnis dieser zueinander zu begreifen, ist eng verknüpft mit der Praxis des Programmierens. Das Erlernen einer Programmiersprache über diese Praxis muss dabei gar nicht unmittelbar auf die Spezifikation der Sprache zurückgreifen: Ein Verständnis kann auch anhand von Beispielen, vereinfachten Erklärungen und Ausprobieren zustande kommen. Raymond Turner beschreibt diesen Lernprozess als empirisches Vorgehen:

> „A programmer attempting to learn a programming language does not study the manual, the semantic definition. Instead, she explores the implementation on a particular machine. She carries out some experimentation, runs test programs, compiles fragments etc. until she figures out what the constructs of the language do. Learning a language in this way is a practical affair. Moreover, this what programmers require in practice [sic]. Indeed, in order to program a user needs to know what will actually happen on a given physical machine. And this is exactly

111 Slonneger, Kenneth, Kurtz, Barry L.: Formal Syntax and Semantics of Programming Languages. Reading. Addison-Wesley. 1995. S. 507.

what such a practical investigation yields.“ [112]

Turner zeigt jedoch auf, dass ein solches Vorgehen eine Sprachspezifikation nicht ersetzen kann: Als empirisches Wissen über die Sprache kann die Kenntnis der Funktionalität nicht die normativen Aufgaben der Spezifikation erfüllen, also z.B. als Vorgabe für den Compilerbau verwendet werden oder festlegen, wann ein Programm eine korrekte Anwendung der Sprache darstellt. Gleichwohl kann solches empirisches Wissen, auch wenn es unvollständig und falsifizierbar bleibt, zur Konstitution einer Programmsemantik verwendet werden.

Losgelöst von geschriebenen Sprachspezifikationen wird den einzelnen Konstrukten einer Programmiersprache also in diesem Fall eine Bedeutung zugeschrieben, die auf ihre Funktion und Anwendung abzielt. Auch Bedeutungszuschreibungen auf einer solchen empirischen Basis sollen hier als maschinelle Semantik verstanden werden. Metaphorisch kann die Bedeutungszuschreibung über die Funktion so beschrieben werden, dass die Sprache als Werkzeugkoffer erscheint, ihre zulässigen Teile entsprechend als Werkzeuge, die für verschiedene Aufgaben verwendet werden können.

In der praktischen Auseinandersetzung mit Computerprogrammen spielen empirische Aussagen über die Semantik eine große Rolle: So sind die meisten Funktionalitätsbeschreibungen zwar abstrakt und normativ – sie beschreiben, was ein Programm tun *soll*. Die Aussage, dass ein Programm diesen Vorgaben entspricht, ist jedoch empirisch (von wenigen Fällen formal verifizierter Programme abgesehen). Und tatsächlich werden die empirischen Aussagen experimentell überprüft: Jeder fehlgeschlagene Test, jeder gefundene Bug, falsifiziert die empirische Annahmen über die Semantik (oder die empirischen Annahmen über die Laufzeitumgebung). Die Semantik ist dabei zwar nicht prinzipiell unbekannt und müsste erforscht werden, da der Quellcode des betreffenden Programms ja zugänglich ist. Allerdings ist sie komplex genug, dass sich die Aussagen ohne Rückgriff auf Experimente nicht einfach durch einen Verweis auf den Code überprüfen lassen.

Im letzten Unterkapitel wurde die Anwendung optimierender Compiler kurz vorgestellt. Neben diesen automatisierten Optimierungen werden an Programmen häufig manuell Optimierungen durchgeführt um das zeitliche Verhalten zu verbessern oder den Ressourcenverbrauch (z.B. von Speicherplatz) zu senken. Wird durch Optimierungen die Semantik von Programmen geändert? Zunächst einmal soll durch Optimierung eine Verbesserung beschrieben werden, die an der Funktionalität des Programms nichts ändert. Zum Beispiel sollen die gleichen Eingaben weiterhin zu den gleichen Ausgaben führen. Aus diesem Grund kann es gerade als Merkmal der Optimierung angesehen werden, dass die Programmsemantik durch diese nicht geändert wird. Allerdings trifft dies nicht auf jede Form von Semantik zu: So kann eine Semantik (auch eine formale) z.B. zeitliches Verhalten oder die Menge an verwendetem Speicher mit erfassen. In diesem Fall wäre es gerade das Ziel der Optimierung, die Semantik im Hinblick auf diese Eigenschaften zu verändern. Selbst ohne Formalisierung von Zeit und Speicherplatz wird die Semantik durch viele Optimierungen geändert: Wird z.B. die Instruktionsreihenfolge angepasst, um *Pipeline Stalls*[113] zu verhindern, kommt dem Programm eine andere operationelle Semantik zu. Wird die Division durch eine

112 Turner, Raymond: Programming Languages as Mathematical Theories. In: Vallverdú, Jordi: Thinking Machines and the Philosophy of Computer Science: Concepts and Principles. 2010. S. 74.

113 Eine Verzögerung der Ausführung des Programms, um Fehler bei der parallelen Bearbeitung mehrerer aufeinander folgender Instruktionen durch den Prozessor zu vermeiden

konstante Zweierpotenz durch ein *Right Shift*[114] ersetzt, so ändert dies die Semantik, auch wenn das Resultat äquivalent ist. Manuelle Optimierungen können noch größere Änderungen zur Folge haben, etwa wenn Daten in einer sparsameren Form festgehalten werden. Betrachtet man das Programm dagegen als deutlich abstrahierte denotationelle Semantik, z.B. als Funktion von Eingaben auf Ausgaben, so haben die genannten Optimierungen keinen Einfluss auf die Semantik – noch deutlicher wird dies in axiomatischer Semantik, wenn die Assertionen durch die Optimierungen nicht berührt sind. Für den Programmbegriff ist festzuhalten, dass eine Form der Semantik gefunden werden kann, in der die Optimierungen zu keinen Änderungen führen, die Semantik also gezielt bestimmte Aspekte des Programms erfassen kann.

Zusammenfassen lassen sich die unterschiedlichen Formen maschineller Semantik als Beschreibung dessen, was eine abstrakte Maschine tut, die das Programm ausführt. Dies unterscheidet sich von der Ausführung auf konkreten physischen Maschinen: Letzterer kann Zeit und Ort zugeschrieben werden, das Programm zeigt mit einer bestimmten Eingabe ein bestimmtes Verhalten. Die maschinelle Semantik eines Programms umfasst das Verhalten bei allen möglichen Eingaben, sie kann nicht verortet werden, sie hängt nicht von einer physischen Realisierung ab. Eine konkrete Programmausführung kann zwar appräsentiert sein, sie kann (z.B. bei Tests) zur Überprüfung von empirischen Annahmen über die Semantik herangezogen werden, sie kann auch zur Beschreibung der Semantik verwendet werden. Sie ist jedoch klar unterschieden von der (formal-) sprachlichen Bedeutung, die dem Programm zugeschrieben wird.

114 Eine Verschiebung der Bits einer binär gespeicherten Zahl. Im Ergebnis multipliziert *Left Shift* mit einer Zweierpotenz, *Right Shift* teilt durch diese. Die Laufzeit kann unter bestimmten Voraussetzungen kürzer sein, als eine durchgeführte Multiplikation bzw. Division.

3.3.2 Natürlichsprachliche Semantik

Neben der Aufgabe, das Verhalten abstrakter Maschinen auszudrücken, bringen insbesondere höhere Programmiersprachen dieses Verhalten auch in eine für Menschen lesbare und verständliche Form. Maschinencode hält nur die Instruktionen fest, die das Verhalten einer abstrakten Maschine definieren oder von einem konkreten Prozessor ausgeführt werden können. Höhere Programmiersprachen abstrahieren vom Befehlssatz eines bestimmten Prozessors und ermöglichen Menschen, Maschinenverhalten in einer deutlich zugänglicheren Sprache auszudrücken.

Ein wichtiges Element, das diese erhöhte Zugänglichkeit ermöglicht, ist die Einbindung natürlicher Sprache in die Ausdrucksmöglichkeiten der Programmiersprachen. Für die Konstitution des Computerprogramms als menschenlesbarer Gegenstand ist insbesondere das Verhältnis der maschinellen Semantik zur Semantik der Elemente natürlicher Sprache, die im Programm auftauchen, entscheidend. Um dieses Verhältnis näher zu beleuchten soll hier zunächst dargestellt werden, an welchen Stellen und in welcher Form natürlichsprachliche Elemente in Computerprogrammen wahrgenommen werden können.

Zunächst begegnet uns natürliche Sprache im Zusammenhang mit Programmen bereits außerhalb des eigentlichen Quellcodes in vielfältigen Formen. Programme werden mit Namen versehen, diese Namen drücken eine Zusammengehörigkeit verschiedener Gegenstände aus, die mit dem benannten Programm zusammenhängen. So wird z.B. deutlich gemacht, dass verschiedene Versionen eines Programms mit gleichem Namen trotz Unterschieden Gemeinsamkeiten aufweisen. Für einige Programme existieren komplexe Namensgebungssysteme, mit denen Gemeinsamkeiten und Unterschiede verschiedener benannter Programme ausgedrückt werden. Ein Beispiel sind die verschiedenen Namen, die im Umfeld von der Linux-Distribution *Ubuntu* verwendet werden:

Ubuntu wird regelmäßig aktualisiert. Die verschiedenen Versionen heißen weiterhin Ubuntu, dadurch wird unter anderem ein gleichbleibendes Ziel bei der Auswahl von Software, die zur Distribution zusammengestellt wird, signalisiert. Darüber hinaus werden größere Unterversionen zur Unterscheidung mit eigenen Namen versehen, die aus einer Alliteration eines Adjektivs mit einem Tier oder Fabelwesen besteht. Die Anfangsbuchstaben sind dabei nach dem Alphabet geordnet: Auf die Version *Zesty Zapus* folgte *Artful Aardvark*, darauf *Bionic Beaver* und *Cosmic Cuttlefish*. Weiter gibt es drei verschiedene Editionen der Distribution für verschiedene Zielsysteme, die Ubuntu Desktop, Ubuntu Server und Ubuntu Core genannt werden[115]. Versionen, in denen die grafische Benutzerumgebung ausgetauscht wurde, existieren z.B. unter den Namen Xubuntu, Lubuntu oder Ubuntu MATE[116].

Diese Namensgebung verdeutlicht, dass die gegenseitigen Verweise und Anspielungen in den Namen über die Unterscheidbarkeit verschiedener Programme hinaus bereits Information vermitteln soll. In vielen Fällen deuten Namen auch bereits die Ausrichtung der Funktionalität eines Programms an, z.B. verweisen *Libreoffice Writer* oder *Microsoft Word* bereits auf die Textverarbeitung, *Windows* betont die Fenster einer grafischen Benutzeroberfläche, GNOME *Calculator* ist sowohl namentlich als auch funktional als Taschenrechner konzipiert. Analog zu diesen Produktnamen drücken auch Dateinamen häufig in natürlicher Sprache bereits Information über den Inhalt aus.

115 Siehe https://www.ubuntu.com/about (abgerufen am 21.08.2018).

116 Siehe https://wiki.ubuntu.com/UbuntuFlavors (abgerufen am 21.08.2018).

Auch Dokumentationen und Anwendungshinweise greifen in hohem Maße auf natürliche Sprache zurück, sei es in Anleitungen, Dokumentationswikis oder auch Schulungen zu einem bestimmten Programm. In diesen Fällen wird die Funktionalität eines Programms, oft erheblich abstrahiert und mithilfe natürlicher Sprache vermittelt, um zur Anwendung vorausgesetztes Wissen weiterzugeben. Der Zugang zur Funktionalität ist hier ein anderer als im Quellcode, da das Maschinenverhalten auf die Aspekte reduziert dargestellt wird, die in der Benutzung des Programms von Bedeutung sind. Eine weitere natürlichsprachliche Quelle, in der die Bedingungen einer Anwendung behandelt werden, sind Lizenzvereinbarung. In diesen werden insbesondere rechtliche Aspekte der Programmanwendung aufgegriffen.

Neben der Dokumentation kann natürliche Sprache in der Interaktion zwischen Anwender_innen und Programmen zur Laufzeit verwendet werden. So können z.B. in Benutzeroberflächen, die über Kommandozeilen bedient werden, Anlehnungen an und Abkürzungen von englischen Wörtern als Instruktionen implementiert sein: In MS-DOS beispielsweise wechselt *cd* (change directory) das Verzeichnis, *copy* kopiert eine Datei. Auch in grafischen Benutzeroberflächen wird natürliche Sprache, beispielsweise in Beschriftungen oder Hinweisfenstern, weiterhin verwendet. Die Informationsvermittlung geht dabei in beide Richtungen: Ein Hinweisfenster oder eine Beschriftung soll als *Output* vom Programm ausgehend Information vermitteln, die Eingabe eines Zielbahnhofs an einem Fahrkartenautomat fungiert als natürlichsprachlicher *Input*.

In diesen Beispielen wird natürliche Sprache zwar im Umfeld von Programmen verwendet, die sprachlichen Gegenstände werden jedoch nicht *als Programm* konstituiert. Im engen Sinne soll hier unter dem Programm als natürlichsprachlicher Semantik daher die Semantik derjenigen Sprache verstanden werden, die im Quellcode des Programms verwendet wird. Eine breite Anwendung natürlicher Sprache im Quellcode und die Verflechtung der maschinellen Semantik mit natürlichsprachlicher Semantik sind dabei ein Spezifikum höherer Programmiersprachen.

Zunächst haben viele höhere Programmiersprachen Schlüsselworte, die natürlicher Sprache (insbesondere der englischen Sprache) entlehnt sind. Die programmiersprachlichen Konzepte, für die diese Schlüsselwörter stehen, sind dabei üblicherweise so aufgebaut, dass sie zumindest teilweise in Übereinstimmung mit der natürlichsprachlichen Verwendung der Worte stehen: Eine „*repeat (…) until (…)*“-Schleife in Pascal verhält sich in der Ausführung ähnlich, wie es die Ausführung einer natürlichsprachlichen Anweisung zur Wiederholung, etwa in einem Kochrezept, erwarten lässt. *void* ist in Java entsprechend der natürlichsprachlichen Bedeutung ein Verweis auf die Leere, z.B. wenn eine Methode keinen Rückgabewert hat. *friend* beschreibt in C++ ein Verhältnis von Klassen und Funktionen, in denen „befreundete“ Klassen und Funktionen auf eigentlich geschützte Bereiche zugreifen können, eine Übertragung des natürlichsprachlichen Konzepts des Vertrauens in eine Programmiersprache. *true* und *false* sind in vielen Programmiersprachen als Literale der Logik entlehnt und zumindest mittelbar mit der natürlichsprachlichen Verwendung von wahr und falsch verknüpft. Diese Verweise auf natürliche Sprache führen zu einer erheblich verbesserten Lesbarkeit von Programmteilen und vereinfachen das Erlernen der Konzepte einer Sprache. Sie sind nicht beliebig: Als Beispiel kann die Vorstellung herangezogen werden, dass ein Programm deutlich schwieriger zu lesen wäre, wenn *true* und *false* durch *a* und *b* ersetzt werden – oder durch *false* und *true*.

Eine große Menge an programmiersprachlichen Konzepten, die natürliche Sprache zulassen, sind alle Formen von Bezeichnern. Mit Bezeichnern werden Variablen, Funktionen, Klassen usw. benannt. In den meisten Sprachen können diese Namen relativ frei als Folge von Buchstaben, Ziffern und einigen Sonderzeichen gewählt werden. Die Namensgebung durch

Bezeichner ist dabei in hohem Maße von den Konzepten der verwendeten Programmiersprache abhängig: Klassen und Interfaces sind mit objektorientierten Sprachen verbunden, die Frage nach ihrer Benennung stellt sich nur dort. Methoden, die innerhalb von Klassen definiert werden und dadurch auf diese bezogen sind, werden in einem anderen Kontext benannt als globale Methoden. Funktionen in funktionalen Sprachen erfüllen eine wesentlich andere Aufgabe als Prozeduren in imperativen Sprachen; diese unterschiedliche Zielsetzung schlägt sich auch in der Benennung nieder. Wie diese Freiheit der Namensvergabe verwendet wird, kann als Frage des Programmierstils angesehen werden: Hierzu gibt es sprachabhängig Hinweise, Konventionen und Debatten über die sinnvolle Verwendung und die Bezüge, die durch die Namen hergestellt werden.

Eine weitere Möglichkeit der Benutzung natürlichsprachlicher Ausdrücke im Quellcode sind die Ein- und Ausgabe. Dies betrifft offensichtlich die Kommunikation mit Menschen, die das Programm anwenden. Allerdings kann natürliche Sprache auch beim Schreiben und Lesen von Daten angewendet werden. Bei der Kommunikation mit Menschen ist der größte Vorteil, dass zur Benutzung des Programms keine weitere Sprache erlernt werden muss. Enthält die grafische Benutzeroberfläche beispielsweise Buttons mit den Beschriftungen *save*, *load*, *forward* usw. ist die Bedienung leichter zu erlernen, als diejenige eines vergleichbaren Programms mit den Beschriftungen *f1*, *f2*, *f3*, … Auch im Beispiel des Fahrkartenautomats ist es intuitiver, den Namen des Zielbahnhofs einzugeben, als diesen über eine Zahl auszuwählen. Dafür müssen die erkannten sprachlichen Elemente im Quellcode (oder einer gelesenen Datei) definiert sein. Dieses Vorkommen natürlicher Sprache im Quellcode ist die Entsprechung zur oben erwähnten Verwendung zur Laufzeit. Das Laufzeitverhalten ist dabei den Codeelementen zur Ein- und Ausgabe appräsentiert. Beim Schreiben und Lesen von Daten kann natürliche Sprache verwendet werden, um beispielsweise menschenlesbare Logfiles zu produzieren. Weiter stellt die Speicherung als einfache Zeichenfolge eine Absicherung gegen Obsoleszenz der Daten dar und ermöglicht die Verwendung vieler Werkzeuge, die auf die Bearbeitung von Text ausgerichtet sind – z.B. die Verwaltung durch Versionskontrollsysteme[117].

Schließlich greifen echte Kommentare (siehe Kapitel 3.2) in hohem Maße auf natürliche Sprache zurück. In diesen wird zusätzliche Information zum ausführbaren Programm selbst vermittelt. Dabei ist es eine Frage des Programmierstils, ob das Verhalten des Programms selbst (z.B. in stark abstrahierter Form) noch einmal in natürlicher Sprache ausgedrückt wird. Andrew Hunt und David Thomas lehnen eine solche Wiederholung in ihrem Buch zu Softwaretechnik *The Pragmatic Programmer* ab, da sie das DRY-Prinzip (Don‘t Repeat Yourself) verletzen würde. Kommentare sollen vorrangig zur Begründung, nicht zur Wiederholung dienen:

> „Producing formatted documents from the comments and declarations in source code is fairly straightforward, but first we have to ensure that we actually *have* comments in the code. Code should have comments, but too many comments can be just as bad as too few.
>
> In general, comments should discuss *why* something is done, its purpose and its goal. The code already shows *how* it is done, so commenting on this is redundant —and is a violation of the DRY principle.
>
> Commenting source code gives you the perfect opportunity to document those

117 Vgl. Hunt, Andrew and Thomas, David: The Pragmatic Programmer. From Journeyman to Master. Addison-Wesley. 1999. S. 73ff.

> elusive bits of a project that can't be documented anywhere else: engineering trade-offs, why decisions were made, what other alternatives were discarded, and so on."[118]

Diesen verschiedenen Formen, in denen Programme eine natürlichsprachliche Semantik enthalten, kommen in der Programmierpraxis mehrere Aufgaben zu. Eine relevante Eigenschaft für die Konstitution des Gegenstands ist dabei, dass die Elemente des Quellcodes sowohl eine maschinelle Semantik als auch eine natürlichsprachliche Semantik aufweisen. In Programmiersprachen, die natürliche Sprache lediglich in Kommentaren zulassen (z.B. einige Assemblersprachen), gestaltet sich das Verhältnis von natürlichsprachlicher Semantik zu maschineller Semantik anders als in solchen, die eine freie natürlichsprachliche Benennung von Bezeichnern zulassen. Natürlichsprachliche Semantik kann ihre Funktion also nur sprachabhängig erfüllen.

Diese Funktion ist eng mit der Aufgabe des Quellcodes verknüpft, das Verhalten einer abstrakten Maschine nicht nur in einer von Computern verarbeitbaren Form, sondern auch in menschenlesbarer Form festzuhalten. Quellcode wird nicht nur dazu verwendet, Instruktionen von Menschen an eine Maschine weiterzugeben – es ist gleichzeitig eine Sprache, in der Menschen miteinander kommunizieren. Eine solche Kommunikation durch Quellcode findet immer dann statt, wenn ein Mensch den von einem anderen Menschen geschriebenen Code versucht zu lesen und zu verstehen. Durch die natürliche Sprache ist neben dem abstrakten Maschinenverhalten weitere Information im Gegenstand enthalten. Wie oben für Kommentare gefordert, können dies zum Beispiel Begründungen und Ziele einer bestimmten Implementierung sein.

Neben Programmen, die von anderen Menschen verstanden oder weiterentwickelt werden sollen ist diese Form der Kommunikation über Quellcode insbesondere auch in der gemeinsamen Entwicklung in Teams von Bedeutung. Quellcode stellt innerhalb des Teams neben seiner technischen Funktion auch einen ständigen Austausch über (Teil-) Ziele, Vorgehensweisen und die Form dar, in der Wissen im Programm abgebildet wird. Diese Abbildung von Wissen geschieht zum Beispiel über Datentypen: Wenn ein Programm für die Anwendung in einem bestimmten Kontext intendiert ist und für seine Ausführung variable Information über diesen Kontext speichern muss, so wird diese Information in Form von digitalen Daten festgehalten. Dieser Bezug zur Kontextinformation ist nicht Teil der maschinellen Semantik: Typenbezeichnungen von Bauteilen und Straßennamen können beide als Zeichenketten gespeichert und verarbeitet werden; Verständnis des Programms setzt jedoch voraus, zu wissen, auf welche Kontextgegenstände sich diese Zeichenketten beziehen. Diese Bezüge können in natürlichsprachlicher Bedeutung des Quellcodes ausgedrückt sein. Ein abgegrenzter, für ein bestimmtes Programm relevanter Kontext wird als Anwendungsdomäne dieses Programms bezeichnet. Innerhalb der Bezüge zur Anwendungsdomäne findet häufig auch eine Operationalisierung statt, eine Festlegung, wie die Information in die Form digitaler Daten kommt und wie das Programm auf diese Daten zugreifen kann.

Die Kommunikation durch Quellcode muss nicht unbedingt zwischen mehreren Menschen stattfinden. Auch das Verständnis des selbst geschriebenen Codes kann mit zeitlichem Abstand schwierig werden. Natürlichsprachliche Semantik kann auch dazu dienen, später die eigenen Entscheidungen anhand der natürlichsprachlich mitgegebenen Information nachzuvollziehen. Dies spielt insbesondere für die Wartbarkeit von Programmen eine wichtige Rolle: Nur mit einem zumindest teilweise gegebenen Verständnis des Quellcodes können

118 Hunt, Andrew and Thomas, David: The Pragmatic Programmer. From Journeyman to Master. Addison-Wesley. 1999. S. 249.

sinnvolle und zielgerichtete Änderungen vorgenommen werden. Da die meisten Programme hinreichend komplex sind, so dass sich nur wenige Menschen alle relevanten Details des Entwicklungsprozesses für eine spätere Wartung und Anpassung in Erinnerung rufen können, müssen diese Details festgehalten werden. Die Relevanz des Domänenwissens und der Begründungen bestimmter Entscheidungen in der Programmentwicklung für die Arbeit an Programmen zeigt, dass zum Verstehen eines Programms mehr Wissen als die Kenntnis der maschinellen Semantik gehört. Mindestens ein Teil dieses Wissens kann über die natürlichsprachliche Semantik vermittelt werden.

Die Funktion eines Programms ist es häufig, ein Problem oder eine Aufgabe in einer bestimmten Anwendungsdomäne zu lösen. Wie genau an den Entwicklungsprozess herangegangen wird, hängt dabei sowohl vom Wissen über die Anwendungsdomäne als auch von der verwendeten Programmiersprache ab. Dabei wird die Aufgabe aus der Anwendungsdomäne in eine von Maschinen lösbare Form gebracht. Die natürlichsprachliche Semantik kann dabei ausdrücken, wie genau sich das Programm auf diese Aufgabe bezieht. Dadurch wird auch vermittelt, wie die Aufgabe aus Sicht der Programmentwicklung gedacht wird. Dies ist sprachabhängig:

> „Computer languages influence *how* you think about a problem, and how you think about communicating. Every language comes with a list of features—buzzwords such as static versus dynamic typing, early versus late binding, inheritance models (single, multiple, or none)—all of which may suggest or obscure certain solutions. Designing a solution with Lisp in mind will produce different results than a solution based on C-style thinking, and vice versa. Conversely, and we think more importantly, the language of the problem domain may also suggest a programming solution.“[119]

Andrew Hunt und David Thomas plädieren hier für eine Softwareentwicklung nahe an der Anwendungsdomäne und für die Entwicklung eigener, auf die Domäne bezogener „Mini-Sprachen“. Domänenspezifische formale Sprachen werden in vielen Anwendungsbereichen verwendet, sie lassen eine formale Fassung der Aufgabenstellung möglichst nahe an der in der Domäne verwendeten natürlichen Sprache zu. Zu diesem Zweck verzichten domänenspezifische Sprachen häufig auf die universelle Anwendbarkeit anderer Programmiersprachen.

Tatsächlich muss für eine Entwicklung, die sich an der Sprache der Anwendungsdomäne orientiert, nicht unbedingt auf eine neue Sprache mit eigener Syntax zurückgegriffen werden. Bereits die übliche Verwendung von Bezeichnern für Variablen, Methoden und Datenstrukturen lassen eine Formulierung von Aufgaben und Lösungsschritten in natürlichsprachlichen Ausdrücken zu. Die entsprechenden Deklarationen stellen eine Erweiterung der Ausdrücke dar: Wenn in der Anwendungsdomäne ein relevanter Gegenstand ausgemacht wird, so kann dieser Gegenstand über einen Bezeichner im Programm deklariert werden, und durch die Deklaration kann sich innerhalb der Programmiersprache auf den Gegenstand bezogen werden.

Als Illustration für diesen Vorgang und für die Relevanz natürlicher Sprache dabei kann die Vorstellung eines Programms dienen, das für einen nicht näher bestimmten Zweck geometrische Formen auf einer zweidimensionalen Ebene verarbeitet. Da hierbei Punkte auf der Ebene eine Rolle spielen, wird an einer Stelle des Programms eine Datenstruktur (oder ein Objekt) „Punkt“ deklariert, der Punkt wird im Programm durch zwei Fließkommazahlen auf der Ebene verortet. Die Deklaration erweitert die Ausdrucksfähigkeit des restlichen

119 Hunt, Andrew and Thomas, David: The Pragmatic Programmer. From Journeyman to Master. Addison-Wesley. 1999. S. 57.

Programms um die Bezeichnung „Punkt“. Dadurch wird die Ausdrucksfähigkeit des Programms um einen domänenspezifischen Begriff erweitert, „Punkt“ erhält durch die Deklaration eine Bedeutung in der maschinellen Semantik, die sich an der domänenspezifischen natürlichsprachlichen Bedeutung orientiert.

Innerhalb dieser Deklaration könnten nun die beiden Fließkommazahlen als „x, y“ oder als „rho, phi“ benannt sein. Die maschinelle Semantik bleibt dabei gleich, da in beiden Fällen zwei Fließkommazahlen benötigt werden. Im ersten Fall verweist der natürlichsprachliche Ausdruck durch Konvention auf kartesische Koordinaten, im zweiten Fall auf Polarkoordinaten. Trotz gleicher maschineller Semantik sollte sich das restliche Programm deutlich unterscheiden, je nachdem, welches Paar von Bezeichnern gewählt wird. Streng genommen kann das Programm in diesen Fällen durch die begrenzte Genauigkeit von Fließkommazahlen nicht einmal die gleiche Menge an Punkten darstellen, abgesehen davon müssten sich in jedem Fall Berechnungen im Bezug auf die Koordinaten unterscheiden. Dies zeigt, dass die natürlichsprachliche Semantik als Form, über eine Aufgabe zu denken, maßgeblich die weitere Softwareentwicklung strukturiert.

3.4 Das Programm als eingebetteter Gegenstand

3.4.1 Das Programm zur Laufzeit

Die letzten beiden Unterkapitel (3.2 und 3.3) haben die Bedeutungsdimensionen entwickelt, die das Programm als syntaktischen und semantischen Gegenstand begreifen. Sie zielen insbesondere auf den Quellcode ab. Die hier entwickelte Bedeutungsdimension, die das Programm als eingebetteten Gegenstand beschreibt, ist für die Anwendungsperspektive von großer Bedeutung.

Die vierte Form, in der Computerprogramme als Gegenstand konstituiert werden können, ist eine Form, die durch den unmittelbaren Bezug zu weiteren Gegenständen maßgeblich bestimmt ist. Das Programm tritt hier als Gegenstand auf, der nicht losgelöst von seinem Kontext gedacht werden kann. Da dieser Kontext – bestehend aus den weiteren Gegenständen, mit denen das Programm im Verhältnis steht – die Bedingungen und Grenzen der Gegenstandskonstitution festlegt, wird das Programm hier in dieser Bedeutung als eingebetteter Gegenstand bezeichnet. Die erste Menge an Gegenständen, die in dieser Form als eingebettet konstituiert werden, sind Programme zu ihrer Laufzeit.

Während die maschinelle Semantik das Verhalten einer abstrakten Maschine beschreibt, wird das Programm zur Laufzeit als Steuerung einer konkreten Maschine konstituiert. Diese konkrete Maschine enthält in den meisten Fällen mindestens einen Prozessor, der eine konkrete Menge von Instruktionen, seinen Befehlssatz, ausführen kann. Die Ausnahme hierzu stellt programmierte Hardware dar – in diesem Fall wird das Programm vor seiner Ausführung nicht in Instruktionen für einen Prozessor übersetzt, sondern in einen Schaltkreis, der das in der maschinellen Syntax beschriebene Maschinenverhalten umsetzt.

Damit Prozessoren die Instruktionen von Programmen zur Laufzeit ausführen können, müssen diese in passender Maschinensprache vorliegen, also im Befehlssatz des Prozessors ausgedrückt sein. Dazu muss das syntaktische Programm vor einer Ausführung in die jeweilige Maschinensprache übersetzt werden oder zur Laufzeit von einem Interpreter überführt werden. Virtuelle Maschinen, die Programme in einer Zwischendarstellung ausführen können, agieren dabei als Compiler bzw. Interpreter (je nach Implementierung) für diese Zwischendarstellung und überführen sie in Maschinensprache.

Während sich ein Programm zur Laufzeit zwar als physischer Vorgang begreifen lässt, scheitert jede Beschreibung auf dieser Ebene an der Komplexität: Selbst auf der abstrahierten Beschreibungsebene von Transistoren oder Gattern können die Aktionen zur Laufzeit eines Programms auf einem aktuellen Prozessor mit mehreren Milliarden Transistoren und mehreren Milliarden Taktzyklen pro Sekunde ohne weitere Reduktionen nicht erfasst werden. Auch die Konstitution von konkreten Programmen zur Laufzeit greift also in erheblichem Ausmaß auf Abstraktion zurück, obwohl der Gegenstand als konkretes Maschinenverhalten erfahren wird. Das Erfassen von laufenden Programmen kann also nur durch eine Reduktion möglicher Information über die Ausführung stattfinden.

In der Anwendungsperspektive ergibt sich diese Reduktion aus der Beschränkung auf die Information, die aufgrund von Eingabe, Ausgabe und externer Dokumentation vorliegt: Das Programm wird in erster Linie über seine Benutzeroberfläche konstituiert – unter Rückgriff auf weitere Information, die über die Funktionalität bekannt ist, etwa durch das

Lesen einer Beschreibung oder des Handbuchs. Je nach Zweck des Programms kann auch die Interaktion mit anderen Programmen oder Sensoren und Aktoren eine Rolle spielen: Bei der Anwendung eines Fahrkartenautomaten ist in den meisten Fällen kaum etwas über das auf der Steuerelektronik ausgeführte Programm bekannt, durch den Kontext wird jedoch nach Wahl der Fahrkarte und Bezahlung erwartet, dass ein Drucker angesteuert wird, der eben diese Fahrkarte erstellt. Dieses Beispiel illustriert auch, mit welcher reduzierten Information ein Programm konstituiert werden kann: Die Anwender_innen des Fahrkartenautomats kennen weder Details über den Quellcode noch wissen sie genaueres über die konkrete Maschine, auf der das Programm läuft. Trotzdem ist dabei bekannt, dass das bediente Programm auf einer konkreten Maschine läuft (die sich üblicherweise innerhalb des Automaten befindet), auf welche Eingaben welche Ausgaben folgen, und dass dem Programm in seiner Interaktion eine Vermittlungsrolle zwischen mehreren Akteuren zukommt – etwa zwischen Reisenden, Banken und Verkehrsunternehmen.

Die Konstitution eines Programms zur Laufzeit kann jedoch auch sehr viel detaillierter erfolgen als in diesem einfachen Beispiel. Um Fehler zu beheben, Programme weiterzuentwickeln oder laufende Programme zu analysieren (z.B. bezüglich der Sicherheit) muss das Verhalten der konkreten Maschine in einem anderen Zugang erfasst werden als über die Benutzeroberfläche. Ein solcher Zugang kann beispielsweise über die vom Prozessor ausgeführten Instruktionen erfolgen.

Um die Ausführung des Programms als Abarbeiten von Instruktionen zu konstituieren, müssen diese Instruktionen in irgendeiner Form festgehalten, gespeichert sein. In den meisten heute verfügbaren Computern werden Instruktionen und Daten im gleichen Speicher festgehalten. Diese Eigenschaft wird im Bezug zu John von Neumanns Arbeit am EDVAC (Electronic Discrete Variable Automatic Computer) als relevanter Teil der Von-Neumann-Architektur bezeichnet:

> „The key advance was the realisation that instructions that control the computer and the data they operate on are both essentially numbers. This is easy to see now: we could for example associate the number 1 with the addition operation and easily store this in memory alongside the data values, the meaning of 1 becoming a simple matter of interpretation. There is certainly some prior art to the ideas of von Neumann, for example Konrad Zuse described the concept circa 1936 in a patent application for his range of early computers and Alan Turing described a complete stored program computer, the Pilot ACE, circa 1946. However, the successful implementation of the so-called **von Neumann architecture** or **stored program architecture** in the EDVAC has become a milestone innovation.“[120]

Dieser Ansatz unterscheidet sich von der Harvard-Architektur, benannt nach dem Harvard Mark 1 Computer, in dem Daten und Instruktionen separat gespeichert[121] und dem Prozessor auf unterschiedlichen Wegen zugeführt werden. In beiden Fällen werden die Instruktionen nacheinander abgearbeitet, ein Takt gibt dabei vor, in welchem Zeitraum (Zyklus) jeweils eine Instruktion bearbeitet wird. In der Von-Neumann-Architektur wird die Position im Programm, die momentan vom Prozessor verarbeitet wird, in einem Progammzähler festgehalten, der die Speicheradresse angibt, von dem die ausgeführte Instruktion gelesen wird. Dadurch sind Sprünge und Schleifen vergleichsweise einfach zu

120 Page, Daniel: A Practical Introduction to Computer Architecture. Springer, 2009. S. 150ff.

121 Siehe ebda. S. 148.

realisieren, da sie durch eine Anpassung des Programmzählers vorgenommen werden können[122].

Auf konkrete, gegenwärtig verwendete Prozessoren bezogen stellt die schrittweise Abarbeitung von Instruktionen bereits eine Vereinfachung der tatsächlichen Vorgänge dar: Um höhere Taktraten zu erreichen werden die Prozessoren so konstruiert, dass sie innerhalb eines Zyklus eine Instruktion nicht vollständig bearbeiten können. Durch die Aufteilung der Instruktionsbearbeitung auf mehrere Taktzyklen kann jeweils die maximal benötigte Signallaufzeit innerhalb eines Zyklus (die Laufzeit auf dem längsten möglichen Weg, dem *kritischen Pfad*) verkürzt werden. Im Ergebnis kann dadurch die Taktfrequenz erhöht werden, während jede einzelne Instruktion mehrere Taktzyklen benötigt. Um trotzdem eine Instruktion pro Zyklus bearbeiten zu können, werden die einzelnen Schritte der Verarbeitung parallelisiert, die Verarbeitung erfolgt in einer *Pipeline*[123]. In der Praxis werden also mehrere Instruktionen gleichzeitig verarbeitet. Dieses Vorgehen der Prozessoren geschieht jedoch weitestgehend für die Softwareentwicklung unsichtbar: Aus Sicht der Software wird eine Instruktion nach der anderen abgearbeitet, Probleme mit der parallelen Verarbeitung werden innerhalb des Prozessordesigns abgefangen und behoben. Das zeitliche Verhalten eines Prozessors ist jedoch dadurch schwierig abzuschätzen[124].

Ein Prozessor, der Instruktionen abarbeitet, stellt den einfachsten Aufbau einer Einbettung des laufenden Programms dar. In allen Verwendungszusammenhängen kommen zu diesem Aufbau weitere Elemente hinzu, so dass das Programm zur Laufzeit eine Funktion erfüllen kann. Bei Programmen auf eingebetteten Systemen interagieren Sensoren und Aktoren mit dem Programm. Auf PCs, Laptops, Smartphones usw. läuft das Programm in einem System, das die gesamte verbaute Hardware, ein Betriebssystem, weitere Programme und Peripherie (z.B. Ein- und Ausgabegeräte) umfasst. Der konstituierte Gegenstand ist dabei in unterschiedlichem Ausmaß von den Teilen des Systems, auf dem das Programm läuft, abhängig. So laufen Programme in vielen Fällen auch nach einem Entfernen von Eingabegeräten weiter (auch wenn sie dann nicht mehr gesteuert werden können), ohne ein Betriebssystem dagegen würden viele Programme gar nicht in Laufzeit existieren können.

Die Einbettung eines laufenden Programms in ein Betriebssystem stellt in diesen Fällen die Voraussetzung dafür dar, dass das Programm auf (flüchtigen) Speicher zugreift, Dateien auf permanentem Speicher lesen und schreiben kann, die restliche Hardware des Computers verwendet, mit anderen Programmen interagiert und über Peripheriegeräte Ein- und Ausgabe zulässt. Innerhalb des Betriebssystems ist das Programm als Prozess (oder *Task*) repräsentiert. Ein Prozess umfasst neben dem ausführbaren Maschinencode den momentanen Zustand eines laufenden Programms, also den Wert des oben genannten Programmzählers sowie weitere Speicherinhalte (z.B. Variablenwerte). Fast alle gegenwärtig verwendeten Betriebssysteme verwalten eine parallele Ausführung mehrerer Programme als

122 Heute werden mit Harvard-Architektur auch Systeme bezeichnet, die lediglich die Adressräume von Daten und Instruktionen trennen. In dieser Bedeutung sind die Unterschiede deutlich geringer, ein solches System verwendet auch einen Programmzähler.

123 Zu moderner Rechnerarchitektur siehe Hennessy, John L., und Patterson, David A.: Computer architecture: a quantitative approach. 5th Ed. Elsevier, 2011.

Eine praktische Einführung in Details der Implementierung siehe Harris, David M., und Harris, Sarah L.: Digital design and computer architecture. 2nd Ed. Morgan Kaufmann, 2013.

124 Ein Beispiel für diese Schwierigkeit sind Sprungvorhersagen: Da Instruktionen parallel ausgeführt werden, ist für die auf einen bedingten Sprung folgenden Instruktionen bis zur Auswertung der Bedingung unklar, ob sie ausgeführt werden. Der Prozessor prognostiziert in einer komplexen logischen Schaltung den Ausgang dieser Auswertung und führt je nach Ausgang die direkt folgenden Instruktionen oder diejenigen am Sprungziel spekulativ aus. Ist die Vorhersage falsch, wird das Ergebnis verworfen und einige Taktzyklen bleiben „ungenutzt“. Eine Abschätzung zeitlichen Verhaltens müsste also einbeziehen, wie gut die Sprungvorhersagen sind.

Multiprogrammbetrieb (auch *Multiprocessing* oder *Multitasking* genannt). Eine zentrale Aufgabe des Betriebssystems ist dadurch die Verwaltung von Ressourcen und ihre Zuteilung zu laufenden Programmen. Die zugeteilten Ressourcen gehen dabei von der tatsächlich zur Ausführung verwendeten Prozessorzeit über den von Programmen verwendbaren Speicherbereich und die restliche Hardware bis zu den Peripheriegeräten: Werden z.B. auf einem PC zwei Textverarbeitungsprogramme parallel ausgeführt, so sorgt die Zuteilung der Tastatur dafür, dass getippte Zeichen nur von einem der Programme aufgenommen werden.

Die Zuteilung von Prozessorzeit erfolgt durch das Betriebssystem, indem durch dieses ein Ablaufplan (*Schedule*) erstellt wird. Nach diesem Plan wird den einzelnen Prozessen Zeit zugeteilt, in denen der Prozessor ihre Instruktionen ausführt. Im Rahmen der Planung kann den Prozessen dabei eine unterschiedliche Priorität eingeräumt werden, so dass Prozesse mit höherer Priorität bei der Zuteilung von Zeit bevorzugt werden.

Andrew Tanenbaum und Herbert Bos beschreiben in ihrer Einführung zu Betriebssystemen den Charakter von Prozessen sowie das Wechseln des aktiven Prozesses durch das Betriebssystem anhand einer Analogie zu einem Backvorgang. An dieser Beschreibung wird auch der Unterschied zum hier entwickelten Programmbegriff deutlich: Das von Tanenbaum und Bos benannte Programm entspricht dem hier beschriebenen syntaktischen Programm, der Prozess dem eingebetteten Programm.

> „The difference between a process and a program is subtle, but absolutely crucial. An analogy may help you here. Consider a culinary-minded computer scientist who is baking a birthday cake for his young daughter. He has a birthday cake recipe and a kitchen well stocked with all the input: flour, eggs, sugar, extract of vanilla, and so on. In this analogy, the recipe is the program, that is, an algorithm expressed in some suitable notation, the computer scientist is the processor (CPU), and the cake ingredients are the input data. The process is the activity consisting of our baker reading the recipe, fetching the ingredients, and baking the cake.
>
> Now imagine that the computer scientist's son comes running in screaming his head off, saying that he has been stung by a bee. The computer scientist records where he was in the recipe (the state of the current process is saved), gets out a first aid book, and begins following the directions in it. Here we see the processor being switched from one process (baking) to a higher-priority process (administering medical care), each having a different program (recipe versus first aid book). When the bee sting has been taken care of, the computer scientist goes back to his cake, continuing at the point where he left off.
>
> The key idea here is that a process is an activity of some kind. It has a program, input, output, and a state. A single processor may be shared among several processes, with some scheduling algorithm being accustomed to determine when to stop work on one process and service a different one. In contrast, a program is something that may be stored on disk, not doing anything.“[125]

Aus Sicht des Betriebssystems verwendet nur ein Prozess gleichzeitig die Rechenzeit des (Einkern-)Prozessors. In der Analogie ist der Backvorgang für die Zeit der Wundversorgung unterbrochen. Die Zuverlässigkeit moderner Betriebssysteme und ihrer Scheduling-Algorithmen ermöglicht es jedoch, ein laufendes Programm zu konstituieren, ohne auf die Details der Zuteilung von Rechenzeit zurückzugreifen: Das

125 Tanenbaum, Andrew S., Bos, Herbert: Modern Operating Systems. Fourth Edition. Pearson, 2015. S. 87f.

Textverarbeitungsprogramm, mit dem ein Mensch aktiv arbeitet, wird als Gegenstand wahrgenommen, unabhängig davon, wie viele der Milliarden von Prozessorzyklen einer Sekunde ihm tatsächlich zugeteilt werden. Mehrere laufende Programme, die in ein funktionierendes System mit Pseudoparallelismus eingebettet sind, stellen sich als gleichzeitig existierende Gegenstände dar, auch die passiven Prozesse (also solche, denen zu einem Zeitpunkt keine Prozessorzeit zugeteilt ist) sind laufende Programme.

Am Beispiel parallel ausgeführter Prozesse werden zwei Eigenschaften der Bedeutungsdimension eingebetteter Programme deutlich: Erstens ist ihre Konstitution als Gegenstand technisch voraussetzungsreich. Ein einzelnes Programm zur Laufzeit setzt Artefakte wie den Prozessor, aber auch Speicher und Ein- und Ausgabegeräte voraus. Die Konstitution mehrerer gleichzeitig auf demselben Prozessor laufenden Programme kann nur stattfinden, wenn ein Betriebssystem mit funktionierendem Prozess-Scheduler die Zuteilung der Ressource Rechenzeit verwaltet. Nur innerhalb dieser technischen Einbettung können auch mehrere Programme auf einem Rechner miteinander interagieren – wobei diese Interaktion weitere vermittelnde Techniken des Betriebssystems voraussetzt.

Zweitens ist auch ein laufendes Programm von der physischen Hardware abstrahiert: Der Zustand eines Programms, etwa die „Registerinhalte“, bleibt auch dann bestehen, wenn das Programm als passiver Prozess gerade nicht aktiv auf den Prozessor zugreift. Zu dieser Zeit sind die „Registerinhalte“ jedoch gerade nicht in den Registern gespeichert, sondern z.B. im flüchtigen Hauptspeicher festgehalten. „Registerinhalt“ beschreibt also einen Teil des Programmzustands, der physikalisch an unterschiedlichen Orten und mit unterschiedlichen Techniken realisiert werden kann – etwa durch Flip-Flops in Prozessorregistern und durch Kondensatoren im SDRAM des Hauptspeichers, je nachdem ob das Programm als Prozess gerade aktiv ist.

Eine weitere Aufgabe, die das Betriebssystem im Rahmen der Ausführung von Programmen übernimmt, ist die Speicherverwaltung. Da unterschiedliche Programme, die nacheinander oder parallel ausgeführt werden, auf dieselben physisch verfügbaren Ressourcen zurückgreifen, verwaltet das Betriebssystem diese Ressourcen und gestaltet die Form, in der laufende Programme mit ihnen interagieren. Der Hauptspeicher eines Computers nimmt dabei eine zentrale Rolle ein, da Programme hier den größten Teil ihres momentanen Zustandes festhalten. Dem Betriebssystem kommt dabei die Aufgabe zu, den Programmen den physisch vorhandenen Speicher so zur Verfügung zu stellen, dass der Speicherzugriff aus Programmsicht unabhängig von weiteren laufenden Programmen ist, und dass mehrere Programme auf die jeweiligen Speicherinhalte der anderen Programme nicht zugreifen können.

Im Ergebnis abstrahiert das Betriebssystem sowohl vom Speicherort (auf dem physischen Speicher) als auch von der tatsächlichen Größe des Hauptspeichers. Das laufende Programm arbeitet auf einem abstrakten Adressraum (bei Segmentierung auf mehreren abstrakten Adressräumen). Die Speicherverwaltung im Betriebssystem übersetzt in Zusammenarbeit mit der Hardware diese virtuellen Adressen in physische Adressen, die sich direkt auf Zellen im Hauptspeicher beziehen. Dadurch können die tatsächlichen Orte, an denen Daten festgehalten werden, abhängig von der anderweitigen Speicherverwendung für ein Programm unsichtbar variieren. Gleichzeitig kann der virtuelle Adressraum unabhängig von der Größe des tatsächlichen Speichers gewählt werden, wenn das Betriebssystem in der Verwaltung die Möglichkeit hat, Teile des flüchtigen Hauptspeichers im größeren Permanentspeicher (z.B. auf einer Festplatte) auszulagern. Dadurch wird für die Programme von der tatsächlichen Größe des Speichers abstrahiert – allerdings kann sich zeitliches Verhalten ändern, da Lese- und Schreibvorgänge auf Permanentspeichern häufig eine längere

Zeit in Anspruch nehmen.

Neben den Kernfunktionen eines Betriebssystems ist ein laufendes Programm in vielen Fällen auf weitere Funktionalitäten angewiesen, die durch andere Programme zur Verfügung gestellt werden. So wird der Zugriff auf viele Hardwaregeräte eines Computers durch Gerätetreiber gesteuert. Gerätetreiber stellen dabei eine Schnittstelle dar, mit denen das Programm interagieren kann, um indirekt mit der Hardware zu interagieren. So kommunizieren Programme z.B. nicht direkt mit einem Monitor, sondern nutzen stattdessen die Funktionalität eines Grafiktreibers, der eine breite Palette unterschiedlicher Funktionen zur Darstellung grafischer Objekte zur Verfügung stellt. Dieser Grafiktreiber interagiert dann seinerseits wiederum mit der Grafikkarte, die schließlich ein Signal an einem physischen Ausgang produziert, das zu einem Monitor übertragen werden kann. Gerätetreiber abstrahieren dabei von den konkreten Eigenschaften der Hardware. Für das laufende Programm stellt sich eine Tastatur als Funktionalität eines Treibers dar, unabhängig davon, wie der Austausch von Daten zwischen Treiber und Tastatur vonstatten geht oder welche physische Verbindung genutzt wird.

Eine weitere Form von Abstraktion in der Umgebung eines laufenden Programms sind Kommunikationsprotokolle. Diese bestimmen, in welcher Form Programme Daten austauschen können. Protokolle können dabei in Stapeln (*Stacks*) verwendet werden, in denen das Verhältnis der Protokolle zueinander sich in Schichten darstellt (siehe auch Kapitel 3.1.2). Ein Protokoll einer bestimmten Schicht greift jeweils auf die zur Verfügung gestellte Funktionalität der nächstniedrigeren Schicht zurück. Durch diesen Aufbau muss ein Programm, das über das Internet mit einem anderen Programm kommuniziert, die Eigenschaften der physischen Verbindungen nicht beachten. Diese werden in einer niedrigeren Protokollschicht behandelt. Wenn also ein Betriebssystem TCP-Verbindungen (*Transmission Control Protocol*, ein weitverbreitetes Kommunikationsprotokoll, das auf die Funktionalität des nächstniedrigeren *Internet Protocol* zurückgreift) bereitstellt, können Programme über dieses kommunizieren, ohne die darunter liegenden Schichten von Kommunikationsprotokollen zu beachten. Jedes Protokoll abstrahiert dabei von den darunter liegenden Schichten, und ein laufendes Programm, das TCP verwendet, kann nur eingebettet in ein System begriffen werden, das ebendiese Funktionalität zur Verfügung stellt.

Eine virtuelle Maschine wie die bereits beschriebene *Java Virtual Machine* abstrahiert für ein laufendes Programm vom Instruktionssatz des verwendeten Prozessors. Das Programm besteht aus Instruktionen, die zum Instruktionssatz der virtuellen Maschine gehören, im Gegensatz zu Maschinencode, der notwendigerweise aus vom Prozessor ausführbaren Instruktionen besteht. Ein Java-Bytecode-Programm läuft also immer eingebettet in eine Umgebung, die eine solche virtuelle Maschine bereitstellt[126].

Zusammenfassend lässt sich ein Programm zur Laufzeit also als Verhalten einer konkreten Maschine begreifen. Diese konkrete Maschine ist jedoch in praktisch keinem realen Fall lediglich ein Prozessor, der die Instruktionen des Programms ausführt. Vielmehr läuft das Programm eingebettet in ein konkretes System, das eben die Funktionalität bereitstellt, auf die vom Programm zurückgegriffen wird. Aus Sicht des Programms abstrahiert dieses System von der physischen Zeit – indem es diese als Abfolge der jeweiligen Instruktionen darstellt und *Multiprocessing* für das Programm unsichtbar handhabt. Es abstrahiert auch vom physisch vorhandenen Speicher – sowohl auf den tatsächlich verwendeten Speicherort als auch auf die

126 Die virtuelle Maschine muss dazu den Bytecode entweder interpretieren, also immer zur Ausführungszeit in vom Prozessor ausführbare Instruktionen übersetzen, oder im Rahmen einer *Just in Time Compilation* Teile des Programms übersetzen und das Resultat für spätere erneute Ausführungen der jeweiligen Programmteile festhalten. Für das Verständnis eingebetteter Programme sind beide Methoden gleichwertig – in jedem Fall läuft das Programm eingebettet in eine Umgebung, die die jeweilige Zwischendarstellung ausführen kann, und ist ohne eine solche Umgebung nicht als Gegenstand denkbar.

insgesamt verfügbare Größe bezogen. Es abstrahiert über Treiber von der tatsächlichen Hardware und über Protokolle von der stattfindenden Kommunikation in Netzwerken. Schließlich kann eine virtuelle Maschine von der vom Prozessor ausführbaren Sprache abstrahieren. Diese Abstraktionen formen zusammengenommen die Umgebung, in der ein Programm ausgeführt wird, bzw. das System, auf dem eben dieses Programm läuft.

Der Gegenstand eines laufenden Programms kann nicht ohne dieses System gedacht werden. Im Gegensatz zur abstrakten Maschine, deren Verhalten die maschinelle Semantik eines Programms beschreibt, handelt es sich dabei z.B. um einen konkreten PC, der zusammen mit dem Betriebssystem und den weiteren jeweils eingebettet laufenden Programmen die Umgebung darstellt, mit der zusammen wirkend das Programm ausgeführt wird. Das laufende Programm stellt sich dabei als das Verhalten dieses konkreten Gesamtsystems dar.

Dieses System muss dabei nicht notwendigerweise aus einem einzelnen Computer bestehen. Verteilte Anwendungen, die aus mehreren Komponenten bestehen, die auf unterschiedlichen Computern laufen, können als einzelnes Programm aufgefasst werden. Solche Programme werden in unterschiedlichen Architekturen entworfen, weit verbreitet sind *Client-Server* und *Peer-to-Peer*. In einem Peer-to-Peer-Netzwerk läuft das Programm auf mehreren gleichgestellten Computern. Diese können untereinander heterogen sein, in der Architektur nehmen sie jedoch gleichartige funktionale Positionen ein. In Client-Server-Architekturen wird dagegen eine Hierarchie implementiert, in der Clients eine andere Funktionalität als Server aufweisen. In beiden Fällen kann die intendierte Funktionalität des Programms eng verknüpft mit der Verteilung auf mehrere (und oft auch geographisch voneinander entfernte) Teilsysteme sein, die zusammen das System bilden, auf dem das Programm ausgeführt wird. Je nach Perspektive können solche Programme dabei als einzelnes, verteiltes Programm, oder aber als mehrere zusammen arbeitende Programme konstituiert werden. Ein Kommunikationsprogramm für Kurznachrichten, das auf einer Peer-to-Peer-Architektur basiert, kann beispielsweise als einzelner Prozess auf dem jeweils verwendeten Endgerät begriffen werden. In einer anderen Perspektive kann es jedoch auch als verteiltes Programm begriffen werden, das auf den heterogenen Endgeräten zusammen läuft, und so die Kommunikation ermöglicht.

Ein typisches Beispiel solcher verteilten Programme, die sowohl auf einem einzelnen Computer als auch auf mehreren Computern im Netzwerk laufen können, ist das von vielen UNIX- und UNIX-ähnlichen Systemen (insbesondere Linux) verwendete X Window System, das für die Implementierung grafischer Benutzeroberflächen verwendet wird. Der X Client ist dabei die eigentliche Anwendung, deren grafische Benutzeroberfläche über den X Server dargestellt wird. Der X Server kann dabei auf dem gleichen Gerät laufen (z.B. bei der Verwendung von Linux als Desktop-Betriebssystem ohne speziellen Rückgriff auf Netzwerkkommunikation für die Oberfläche), er kann jedoch auch auf einem anderen Endgerät laufen, so dass Client und Server über ein Netzwerk miteinander kommunizieren. Auf diese Art können Terminals betrieben werden, die lediglich den X Server ausführen, und zur Steuerung des Systems mit dem laufenden X Client verwendet werden[127]. Das X Window System kann dabei nur als verteiltes Programm begriffen werden, da sowohl Client als auch Server alleine keine anwendbare Funktionalität bereitstellen.

Im erweiterten Sinne können bei der Konstitution laufender Programme auch Anwender_innen als Teil des Gesamtsystems, welches die Ausführung der Programme

127 Die Architektur des X-Protokolls vertauscht das erwartete Verhältnis von Client und Server miteinander. Zum Beispiel würde der X Client auf einer Workstation mit hoher Rechenleistung laufen, während das minimal ausgestattete Terminal den X Server ausführt. Im www ist das Verhältnis umgekehrt: Die rechenstarkten Webserver verteilen über das http-Protokoll Webseiten, die auf rechenschwachen Endgeräten mit einem Browser gelesen werden können.

ermöglicht, begriffen werden. Viele Programme sind in ihrer Funktionalität direkt auf die Interaktion mit Anwender_innen ausgerichtet. Ein Textverarbeitungsprogramm tut ohne Eingabe gar nichts, und auch die Ausgabe muss über die Verwendung der Benutzeroberfläche gesteuert werden. Bei einer solchen Gegenstandskonstitution ist das Programm in eine Umgebung eingebettet, die auch menschliche Akteure mit einbezieht. Wenn zwei Anwender_innen über ein *Voice-Chat*-Programm miteinander sprechen, so kann das Programm als auf einem System laufend konstituiert werden, dass aus Computern, Netzwerkhardware, weiteren Programmen, externen Sensoren (Mikrofonen) und Aktoren (Lautsprechern) besteht. Im erweiterten Sinne werden die Anwender_innen, die direkt mit den Sensoren und Aktoren in Verbindung stehen, als zum Gesamtsystem dazugehörend verstanden.

Eine solche Gegenstandskonstitution wird in der Softwareentwicklung zum Beispiel bei der Erstellung von Anwendungsfällen (*use cases*) verwendet: Ein Anwendungsfall beschreibt das Verhalten eines Gesamtsystems, das Aktionen von Anwender_innen mit einbezieht. Sie stellen die sprachliche Beschreibung eines Szenarios dar, in denen das entwickelte Programm ausgeführt wird und eine bestimmte Funktionalität erfüllt. Diese Anwendungsfälle sind zwar fiktiv, beschreiben aber das Verhalten eines gedachten konkreten Systems.

Die Einbeziehung von Menschen in Systeme, in denen eingebettet Programme laufen, stellt einen Grenzfall dieser Form der Gegenstandskonstitution dar. In vielen Fällen würden sie eher außerhalb des laufenden Programms und mit diesem interagierend wahrgenommen werden. In einigen Fällen, wie zum Beispiel der Einbettung von Steuersoftware komplexer technischer Systeme oder ganzer Fabriken in ebendieses Umfeld, können jedoch die mit dem technischen System interagierenden Menschen als Teil desselben aufgefasst werden. Andere Beispiele umfassen *Wearables* (also computergesteuerte am Körper getragene Gegenstände) oder medizinische Geräte wie Herzschrittmacher, die ohne mit ihnen unmittelbar in Verbindung stehenden Menschen nichts tun. Häufiger stellen Programme jedoch Bezugspunkte dar, mit denen Menschen in Interaktion treten können. Diese weitere Form eingebetteter Gegenstände wird im nächsten Unterkapitel behandelt.

3.4.2 Das Programm als struktureller Bezugspunkt

Die im letzten Absatz angedeutete Fassung von Programmen als Teil komplexer Systeme stellt einen Perspektivwechsel dar. Programme werden dadurch nicht mehr primär über eine Abgrenzung als Gegenstand konstituiert. Stattdessen werden sie über ihre Wirkungen, die auf sie bezogenen Handlungen und ihre Funktion in Handlungszusammenhängen wahrgenommen. In dieser Perspektive sind Programme auch eingebettete Gegenstände: Sie sind eingebettet in Handlungszusammenhänge, in denen sowohl menschliche als auch nichtmenschliche Akteure eine Rolle spielen. Die Form, in der das Programm dabei als Gegenstand konstituiert wird, ist maßgeblich von seinen Verhältnissen zu anderen Akteuren bestimmt.

Programme zur Laufzeit sind in ein technisches System eingebettet, auf dem sie ausgeführt werden. Die Einbettung in Handlungszusammenhänge führt zu einer Veränderung der Form, in der der Gegenstand Programm wahrgenommen wird. Er stellt nun nicht mehr ein konkretes Maschinenverhalten von Computern dar, sondern ist ein Knoten in einem Netzwerk aus Handlungsträgern, die konkretes Handeln bedingen. Wird zum Beispiel eine Nachricht über einen *Instant-Messaging-Dienst* verschickt, so sind in diese Handlung neben der Intention des versendenden Menschen und seinem Verhältnis zum Empfänger auch die Netzwerktechnik, die verwendeten Computer oder Smartphones und die darauf laufenden Clients des Dienstes als Programme eingebunden. Das Programm wird hier als eines der heterogenen Dinge konstituiert, auf die sich die Handlung des Versendens einer Nachricht bezieht. Da das Programm als Knoten in der Struktur von Handlungsträgern konstituiert wird, wird die Form, in der der Gegenstand Programm dadurch wahrgenommen wird, hier struktureller Bezugspunkt genannt.

Die begriffliche Fassung von Programmen, die in Handlungszusammenhänge eingebettet sind, ist stark an Bruno Latours Akteur-Netzwerk-Theorie (ANT) angelehnt. In dieser sind Handlungen nicht einfach einem Individuum zuzuschreiben, sie entstehen stattdessen in Zusammenhängen, in denen viele Akteure eine Rolle spielen. Akteure können dabei Menschen sein, es können jedoch auch (technische) Gegenstände oder Strukturen sein. Die Form, in der Akteure in einem Handlungszusammenhang beschrieben werden, wird dabei *Figuration* genannt. Ausgangspunkt der Überlegungen ist dabei, dass Handeln nicht losgelöst von den Gegenständen, die bei diesem Handeln eine entscheidende Rolle spielen, untersucht werden kann.

> „Wenn Handeln a priori auf das beschränkt ist, was Menschen »intentional«, »mit Sinn« tun, so ist kaum einzusehen, wie ein Hammer, ein Korb, ein Türschließer, eine Katze, eine Matte, eine Tasse, eine Liste oder ein Etikett handeln könnten. Sie mögen im Bereich »materieller«, »kausaler« Beziehungen existieren, doch nicht im »reflexiven«, »symbolischen« Bereich sozialer Beziehungen. Wenn wir dagegen bei unserer Entscheidung bleiben, von den Kontroversen um Akteure und Handlungsquellen auszugehen, dann ist *jedes Ding*, das eine gegebene Situation verändert, indem es einen Unterschied macht, ein Akteur – oder, wenn es noch keine Figuration hat, ein Aktant. Daher sind die hinsichtlich jeglichem Handlungsträger zu stellenden Fragen einfach die folgenden: Macht er einen Unterschied im Verlauf der Handlung irgendeines anderen Handlungsträgers oder nicht? Gibt es irgendeine Probe, einen Versuch, der es jemandem erlaubt, diesen Unterschied zu ermitteln?

> Die Antwort des Common sense sollte eher ein vernehmliches »Ja« sein. Wenn Sie mit unbewegtem Gesicht behaupten können, daß es genau dieselbe Tätigkeit ist, einen Nagel mit und ohne Hammer einzuschlagen, Wasser mit und ohne einen Wasserkessel zu kochen, Vorräte aufzubewahren mit und ohne einen Korb, durch die Straßen zu gehen mit und ohne Kleider, ein Fernsehgerät mit oder ohne Fernbedienung zu zappen, einen Wagen abzubremsen mit und ohne eine Bremsschwelle, ein Inventar zu führen mit und ohne eine Liste, eine Firma zu betreiben mit und ohne Buchhaltung, daß also die Einführung dieser prosaischen Geräte »nichts wesentliches« an der Durchführung der Aufgaben ändert, dann sind Sie im Begriff, auf den fernen Planeten des Sozialen auszuwandern und aus dieser niederen Welt zu verschwinden. Für all die anderen Gesellschaftsmitglieder macht es einen Unterschied, der unter Erprobung deutlich wird, und so sind diese Geräte, entsprechend unserer Definition, Akteure oder genauer *Beteiligte* am Handlungsverlauf, die darauf warten, eine Figuration zu erhalten.“[128]

Latours Kritik zielt auf die Blindheit vieler soziologischer Theorien gegenüber den (technischen) Gegenständen ab. Handlungen sind nach ihm nicht allein in soziologischen Begriffen erklärbar, wenn diese Begriffe selbst einen wesentlichen Teil des Umfelds, in dem sich Menschen verhalten, ausblenden. In der ANT ist die Handlung eines Menschen, der einen technischen Gegenstand benutzt, weder alleine als intentionale menschliche Handlung, die den Gegenstand ausblendet, noch alleine aus dem Gegenstand heraus erklärbar. Sie kommt durch das Zusammenwirken von Mensch und Gegenstand zustande, daher stehen beide in dem Handlungszusammenhang, der die Handlung hervorbringt.

Die ANT arbeitet sich dabei primär an der Soziologie ab, die das Handeln verkürzt, indem die Bedeutung von Gegenständen nicht hinreichend beachtet wird. Demgegenüber stellt sie einen Handlungsbegriff, in dem die wesentlichen Akteure zunächst unbestimmt sind und erst durch empirische Untersuchung aufgedeckt werden. Auf Programme bezogen kann diese Kritik auch mit umgekehrtem Vorzeichen angewendet werden: Programmen wird häufig ein eigenständiges Verhalten unterstellt, sie „tun“ etwas. Auch dieses Verhalten lässt sich nicht alleine aus dem Programm heraus erklären. Das Programm ist gemacht, seine Funktionalität ist auf eine Umgebung ausgerichtet, in der sein Verhalten auf weitere Gegenstände, andere Programme, Anwender_innen usw. bezogen wird. Erstellt zum Beispiel ein Programm einen Raumbelegungsplan für eine Universität, dann ist das Verhalten des Programms durch die Beschreibung seiner maschinellen Semantik nicht hinreichend erklärt. In das Verhalten sind zahlreiche Bedürfnisse, Erwartungen, Regelungen, organisatorische Vorgaben, Prognosen usw. einbezogen, durch die dieses Verhalten erst erklärbar und relevant wird. In diesem Beispiel kommt dem Programm auch erst dadurch eine Funktion in der Organisation zu, dass sich Menschen nach dem Ergebnis der Planerstellung richten.

Den als strukturelle Bezugspunkte konstituierten Programmen kommen durch ihre Einbettung in Handlungszusammenhänge zwei wesentliche Eigenschaften zu. Erstens „tun“ sie immer mehr, als ihr programmiertes Verhalten zu zeigen. Durch die Einbettung und ihr Verhältnis zu anderen Akteuren strukturieren sie Handlungen mit, die auf den ersten Blick gar nicht von ihnen ausgehen. Zweitens können sie auch in Handlungszusammenhänge eingebettet sein, ohne überhaupt ein Verhalten zu zeigen. Ein Programm kann auch ohne zu einem bestimmten Zeitpunkt auf irgendeinem System zu laufen in Handlungen mit einbezogen sein.

Selbst Programme, die lediglich als „Werkzeug“ gedeutet werden, formen die

128 Latour, Bruno: Eine neue Soziologie für eine neue Gesellschaft. Einführung in die Akteur-Netzwerk-Theorie. Aus dem Englischen von Gustav Roßler. Suhrkamp, 2007. S. 123f.

Handlungszusammenhänge, in die sie einbezogen sind, wesentlich mit: Wird zum Beispiel für ein Softwareprojekt das Versionsverwaltungssystem Git verwendet, so strukturiert die Funktionalität dieses Systems die Perspektive der beteiligten Entwickler_innen auf Versionen der Software mit, und diese Perspektive hat greifbare Auswirkungen auf den Entwicklungsprozess. Die Perspektive ist durch das Programm Git anders strukturiert, als sie es bei der Verwendung des Versionsverwaltungssystems Subversion wäre, und insbesondere ist sie deutlich anders strukturiert, als sie es ohne Versionsverwaltungssystem wäre. Das Werkzeug hat also über seine Funktionalität hinaus Auswirkungen auf die mit der Entwicklung verknüpften Handlungen.

Auch Programme, die gar nicht aktiv auf einem System laufen, treten in Handlungszusammenhängen auf und bestimmen diese mit. Wird zum Beispiel ein bestimmter Instant-Messaging-Dienst verwendet, so signalisiert dies eine prinzipielle Erreichbarkeit über diesen Kanal, auch wenn das Endgerät gar nicht eingeschaltet ist. Der Nachweis einer IT-Sicherheitslücke in einem Labor kann ganze Kaskaden von Handlungen auslösen, von Warnungen über Berichte und Diskussionen bis zu Updates und Vorsorgehandlungen, selbst wenn ein Programm, das diese Sicherheitslücke ausnutzt, tatsächlich nie geschrieben wurde. Alleine die prinzipielle Möglichkeit der Existenz eines Programms mit bestimmter Funktionalität formt dabei Handlungen mit.

Das Beispiel des Instant-Messaging-Dienstes zeigt auf, dass die Verwendung eines bestimmten Programms Aufschluss über die Beziehungen menschlicher Akteure zueinander geben kann: Wenn ein bestimmtes Programm (oder ein Programm, das auf ein bestimmtes Protokoll zurückgreift) verwendet wird, so ist dies gleichbedeutend mit einer Kommunikationsmöglichkeit im Netzwerk, das eben durch dieses Programm oder Protokoll bestimmt ist. Die Programmverwendung ist Bedingung, Anzeichen und Resultat einer Kommunikation mit den anderen Teilnehmer_innen des Netzwerks. Bedingung ist die Verwendung, da sie Kommunikation über die jeweiligen technischen Gegenstände erlaubt. Anzeichen ist die Programmverwendung, da über sie die Einbindung in das jeweilige Netzwerk menschlicher Akteure gesehen werden kann. Und Resultat ist die Verwendung, da die Kommunikationsprogramme in vielen Fällen angewendet werden, um Kommunikation zwischen Menschen, zwischen denen bereits aus anderen Zusammenhängen Kontakte und Beziehungen bestehen, zu erleichtern.

Besonders deutlich wird dieser Handlungszusammenhang, in dem Programme auftauchen, bei großen sozialen Netzwerken. Während viele dieser Dienste ihre Funktionalität über Webseiten anbieten, kann oft auch über Anwendungsprogramme auf Smartphones oder Tablets auf sie zugegriffen werden. Der größte Teil der Programmausführung findet dabei serverseitig statt. Sowohl regional als auch auf die Intention der Anwender_innen bezogen werden unterschiedliche Netzwerke besonders häufig verwendet. Da der Nutzen für die Anwender_innen stark von der Verbreitung des jeweiligen Netzwerks abhängt, können die Netzwerke in ihrer Attraktivität regional relativ stabil sein: Welches soziale Netzwerk für Kommunikation gewählt wird hängt in hohem Maße davon ab, welches Netzwerk andere Menschen, zu denen eine Beziehung besteht, nutzen. Dadurch ist die Einbindung in ein regional besonders attraktives soziales Netzwerk in hohem Maße beeinflusst von der Kommunikation mit Menschen aus der Region und gleichzeitig ein Mittel für weitere Kommunikation. Soziale Bindungen können sich also in diesem Fall auch in der verwendeten Technik ausdrücken.

Programme als strukturelle Bezugspunkte spielen eine besonders wichtige Rolle in den Handlungszusammenhängen im Umfeld großer Organisationen. Programme bestimmen zum Beispiel bei Unternehmen, Universitäten, Regierungen, großen Vereinen usw. die

Organisationsstrukturen mit, sie können mithin als wichtiger Teil der Organisationsstruktur aufgefasst werden. In den Organisationen sind dann viele Handlungen mittelbar und unmittelbar auf das Programm bezogen. Es kann in dieser Einbettung gleichzeitig bestimmte Handlungen ermöglichen und andere verunmöglichen, neue Prozesse erfordern und andere obsolet machen.

Setzt zum Beispiel ein Unternehmen ein bestimmtes Dokumentenmanagementsystem ein, so hat dies Auswirkungen auf zahlreiche Handlungszusammenhänge in diesem Unternehmen, und zwar weit über die Handhabung von Dokumenten hinaus. So kann durch die Einführung im besten Fall auf Dokumente in Papierform teilweise verzichtet werden. Dies hat unter anderem Auswirkungen auf den Einkauf (weniger Papier, dafür evtl. Server- und Netzwerktechnik) und die Abfallwirtschaft (weniger Papier, evtl. mehr defekte oder obsolete Geräte). Die Einführung eines solchen Systems kann die interne Relevanz von IT-Support erhöhen, neue Sicherheitskonzepte notwendig machen, die Raumaufteilung ändern (wenn z.B. ein Archiv nicht mehr benötigt wird, dafür ein neuer Serverraum) und sogar die interne Relevanz ganzer Abteilungen im Unternehmen verschieben. In manchen Fällen könnten im Zusammenhang mit dem System Arbeits- und Werkverträge geschlossen oder aufgehoben werden (z.B. bezogen auf IT-Support, Digitalisierung von Archivbeständen, Klimatechnik).

Neben diesen größeren strukturellen Änderungen werden zahlreiche kleinteilige Handlungen durch das Programm mit strukturiert, von der direkten Handhabung der Dokumente bis zur regelmäßigen Überprüfung der Kühlung im Serverraum. Die Einbettung des Programms in die zahlreichen Zusammenhänge des Unternehmens drückt sich auch sprachlich aus: Der Satz „Unternehmen X verwendet das Dokumentenmanagementsystem Y.“ sagt mehr aus, als „Y läuft auf einem Computer im Unternehmen X.“ Vielmehr wird dadurch ausgedrückt, dass zahlreiche Handlungen in diesem Unternehmen auf Y bezogen sind, mithilfe der Funktionalität des Programms durchgeführt werden und auf die weitere Verwendung von Y ausgerichtet sind.

Ein weiteres Beispiel, in dem Programme große Teile der Handlungszusammenhänge in einer Organisation mit strukturieren sind Campus-Management-Systeme an Hochschulen. Diese Programme verwalten zahlreiche Vorgänge an Hochschulen, von der Einschreibung Studierender über die Handhabung von Prüfungen bis zur Raumzuteilung für Lehrveranstaltungen. Ähnlich dem Beispiel des Dokumentenmanagementsystems werden durch die Einführung, aber auch durch die genaue Ausgestaltung der Funktionalität eines solchen Systems Handlungszusammenhänge maßgeblich verändert, bestimmte Handlungen ermöglicht oder vereinfacht und andere verunmöglicht oder erschwert. Bei Campus-Management-Systemen tritt noch deutlicher hervor, wie das Programm in Zusammenhängen mit anderen Bereichen verknüpft ist und Handlungen formt. Neben den Handlungszusammenhängen, mit denen das Programm direkt in Verbindung mit Studierenden, Lehrenden und Verwaltungsangestellten steht, hängt es zum Beispiel auch über zahlreiche Verknüpfungen mit dem Hochschulrecht zusammen. Das Programm bildet bestimmte rechtliche Anforderungen ab und setzt diese um, gleichzeitig greifen die rechtlichen Institutionen die Verwendung von Programmen für bestimmte Aufgaben auf und müssen diese bewerten, z.B. in Hinsicht auf die notwendige Verfügbarkeit des Systems oder auf die Frage bezogen, ob eine verbindliche Prüfungsanmeldung per Smartphone gleich behandelt wird, wie eine solche durch unterschriebenes Formular.

Parallelen zum Campus-Management-System weisen die verschiedenen Business-Management-Systeme in Unternehmen auf. Ihre Funktionalität umfasst dabei die Abbildung und Verwaltung eines großen Teils der Geschäftsprozesse, die im Unternehmen stattfinden. Sie berühren dabei sämtliche Bereiche des Unternehmens, von der Beschaffung über die

Produktion, Personalwirtschaft und Entwicklung bis zum Verkauf, Rechnungswesen und Controlling. Diese Programme (typischerweise ist eine ganze Reihe von einzelnen Anwendungsprogrammen, Verwaltungssystemen und Datenbanken gleichzeitig im Einsatz) strukturieren dadurch fast alle Handlungszusammenhänge, in denen im Unternehmen Handlungen zustande kommen, mit. Dabei bildet die Software Unternehmensprozesse nicht lediglich ab und ermöglicht sie, stattdessen bestimmt sie maßgeblich die strukturellen Zusammenhänge, in denen Akteure im Unternehmen stehen, mit, und die Akteure richten die Entwicklung von Prozessen an der verwendeten Software aus. Wenn zum Beispiel ein Unternehmen SAP ERP[129] verwendet, so gestaltet die Verwendung der darunter gefassten Programme den Aufbau und die Abläufe im Unternehmen mit. Sämtliche Handlungszusammenhänge im Umfeld der Geschäftsprozesse schließen die Programme mit ein. Die Software stellt dadurch einen wichtigen Bezugspunkt in der Unternehmensstruktur dar.

Schließlich bezieht sich auch die Programmentwicklung in vielen Fällen auf andere Programme. So stellt beispielsweise das Betriebssystem, für das ein bestimmtes Programm entwickelt wird, einen strukturellen Bezugspunkt für die Handlungen im Rahmen dieser Softwareentwicklung dar. Eine Entwicklung für Microsoft Windows gestaltet sich anders als eine Entwicklung für Linuxsysteme, eine Entwicklung für Apple-Computer bringt andere Handlungen hervor als eine angestrebte Plattformunabhängigkeit. Der Bezug auf ein oder mehrere Betriebssysteme kann viele Entscheidungen in der Entwicklung beeinflussen, von der Wahl einer geeigneten Programmiersprache bis zu einer intendierten Kompatibilität zu weiteren Programmen. Auch die Wahl der verwendeten Entwicklungswerkzeuge kann vom System, für das entwickelt wird, abhängen.

Kompatibilität, Interoperabilität und Portabilität sind drei Eigenschaften von Programmen, die sich wesentlich auf andere Programme (oder auch auf Hardware) beziehen. In allen Fällen wird die Softwareentwicklung durch die Existenz anderer Programme (und sei es nur einer früheren Version des entwickelten Programms) maßgeblich beeinflusst. Und auch in diesen Fällen ermöglichen oder verunmöglichen die Programme bestimmte Handlungen, sie strukturieren die Softwareentwicklung mit. So kann zum Beispiel eine erforderliche Rückwärtskompatibilität die Weiterentwicklung eines Programms einschränken, eine angestrebte Portabilität die in der Entwicklung verwendeten Techniken mitbestimmen und die Möglichkeit einer Interoperabilität neue Anwendungsperspektiven für ein entwickeltes Programm eröffnen.

Die Gegenstandskonstitution als struktureller Bezugspunkt greift dabei immer nur einen Teil der Eigenschaften des Programms auf. So können für eine Interoperabilität nur bestimmte Teile der Funktionalität relevant sein, eine Kompatibilität stellt in einer bestimmten Hinsicht eine Gleichwertigkeit dar, ohne die interne Funktionsweise des Programms dabei aufzugreifen. Das entwickelte Programm wird dabei in ein Verhältnis zum Programm, auf das sich die Eigenschaft bezieht, gesetzt. So orientiert sich zum Beispiel ein Textverarbeitungsprogramm, das die Dokumente eines anderen Programms verarbeiten können soll, primär am Format eben dieser Dokumente. Außer dem erzeugten Format muss für eine solche Interoperabilität keine weitere Programmeigenschaft des jeweils anderen Programms beachtet werden[130]. Bei einer plattformübergreifenden Entwicklung muss insbesondere auf die Programmteile geachtet werden, die unter bestimmten Betriebssystemen problematisch sein können und der Portabilität zuwider laufen. Kompatibilität nimmt auf die

129 SAP SE ist der Anbieter einer weltweit verbreiteten Software für Geschäftsprozesse, ERP steht für Enterprise Resource Planning und fungiert als Steuerungssoftware für Abläufe in einem Unternehmen.

130 Weitere Eigenschaften können jedoch aus Gründen der Kompatibilität einbezogen werden.

Schnittstellen bezug, an denen die Gleichwertigkeit des entwickelten Programms angestrebt wird.

In einigen Fällen kann die Programmentwicklung auch in einem negativen Bezug zur Funktionalität anderer Programme stehen. So kann es aus wirtschaftlichen Gründen zum Beispiel sinnvoll sein, Interoperabilität an manchen Stellen zu verhindern, um ein Monopol nicht zu gefährden. Das Schließen einer Sicherheitslücke setzt sich in einen negativen Bezug zu etwaiger Schadsoftware, dadurch soll gerade verhindert werden, dass sie ihre intendierte Funktionalität erfüllen kann. *Digital Rights Management* soll die Nutzung kopiergeschützter Werke beschränken, Software zum Kopieren soll dadurch auf die geschützten Werke nicht angewendet werden können. Kryptografische Software hat als wesentliche Ziele, dass eine Entschlüsselung nur durch bestimmte Personen oder Geräte stattfinden kann, dass sich niemand fälschlicherweise authentifizieren kann, dass Nachrichten nicht verfälscht werden können usw. Damit steht ihre Entwicklung in negativem Bezug zu Software, die dies gerade versucht und umgekehrt.

Programme werden immer dann als struktureller Bezugspunkt konstituiert, wenn sie in Handlungszusammenhängen einen Unterschied machen. Dieser Unterschied kann im Wesentlichen durch die Existenz eines Programms mit dieser Funktion zustande kommen (eine numerische Integration mit dafür geschriebener Software ist maßgeblich etwas anderes als dieselbe Integration mit Papier und Bleistift). Er kann jedoch auch von der genauen Art und Weise abhängen, wie das Programm funktioniert und wie diese Funktionalität präsentiert wird. Und dieser Unterschied kann auch durch weitere Akteure beeinflusst sein: Die Verwendung eines bestimmten Instant-Messaging-Dienstes ist durch die Gruppe der anderen Anwender_innen, die dadurch erreichbar werden, mindestens so stark charakterisiert, wie durch die Details seiner Funktionalität alleine.

3.5 Das Programm als Assoziation

In den vorangehenden Unterkapiteln (3.1 – 3.4) wurde aufgezeigt, dass unter den Programmbegriff sich voneinander stark unterscheidende Gegenstände subsumiert werden. Durch die hier vollzogene analytische Trennung dieser vier Bedeutungsdimensionen kann präzise ausgedrückt werden, welche Form von Gegenstand gemeint ist, wenn von einem Programm gesprochen wird. Das wirft die Frage auf, ob es überhaupt sinnvoll ist, von einem einzelnen Begriff für Computerprogramme auszugehen. Damit ein zusammenhängender Begriff, der die separierten Bedeutungsdimensionen vereint, einen analytischen Mehrwert bietet, muss der Zusammenhang zwischen diesen einzelnen Bedeutungen aufgezeigt werden.

Die Verknüpfung der verschieden konstituierten Gegenstände, die hier unter den Programmbegriff fallen, ist jedoch nicht nur eine Frage der Terminologie. Vielmehr setzt die gesamte Technik im Umfeld von Computern ein Zusammendenken der Gegenstände voraus: Erst durch die Möglichkeit, mittels Übersetzung und Ausführung ein konkretes Maschinenverhalten zu erzeugen, wird ein Datenträger zu einem Träger von Computerprogrammen oder eine Datei zur Programmdatei. Erst durch die Möglichkeit, durch die Interpretation in einer bestimmten Programmiersprache eine Semantik zu erhalten, lässt sich eine Zeichenfolge als Programm auffassen. Erst durch die Verknüpfung mit ausgeführtem Code kann eine sich verhaltende Maschine als Computer oder computergesteuert funktionieren. Die innere Struktur des hier vorgestellten Begriffs ist deshalb nicht lediglich ein Zusammendenken verschiedenartiger Gegenstände, das Verständnis ihrer Verhältnisse zueinander ermöglichen soll. Tatsächlich sind die Zusammenhänge in hohem Maße technisch bedingt und intentional hergestellt, da Computertechnologie ohne sie schlicht nicht existieren würde.

Das Zusammendenken der Gegenstände in den unterschiedlichen Begriffsdimensionen lässt sich zunächst als objektive Appräsentation (Siehe Kapitel 2.2) erfassen. Bei der Wahrnehmung eines Gegenstands sind objektiv appräsentierte Gegenstände solche, die nicht direkt in der Wahrnehmungshandlung konstituiert werden. Stattdessen werden sie indirekt als mitgegeben, mitgegenwärtig konstituiert. Die Rückseite eines gesehenen Gegenstands wird nicht direkt wahrgenommen, jedoch ist sie indirekt in der Wahrnehmung mitgegeben: Sie wird als Teil des Gegenstands konstituiert, auch wenn ihr genaues Aussehen nicht wahrgenommen wird. Objektiv ist diese Form der Appräsentation, da hier die mitgegebenen Objekte von Relevanz sind. Zwar könnten aus der Wahrnehmung eines Programms heraus auch Rückschlüsse über andere Subjekte gezogen werden, etwa über die Intentionen der Entwickler_innen, für den inneren Begriffszusammenhang des hier entwickelten Programmbegriffs sind jedoch gerade die mitgegenwärtigen Programme in jeweils anderen Bedeutungsdimensionen des Begriffs von Interesse – also Objekte.

Die direkt wahrgenommenen Gegenstände sind als physisch konstituiert, sie werden zu einer bestimmten Zeit an einem bestimmten Ort im physisch gedachten Raum wahrgenommen. Dies gilt für greifbare Datenträger, aber auch für die auf Monitoren angezeigten Zeichen, für Schrift auf Papier und für die physisch wahrgenommene Bewegung eines programmgesteuerten Elektromotors in einem Roboter. Dadurch stellen die physischen Programme eine Grundlage zur Wahrnehmung von Programmen in allen Bedeutungsdimensionen dar: Sowohl der Quellcode eines Paketmanagers als auch der Maschinencode eines Betriebssystems, Programmdateien, Robotersteuerungen, ein Beweis über Assertionen bezogen auf die Semantik eines Programms oder die Ausgabe eines Debuggers müssen vermittelt über die physischen menschlichen Sinne wahrgenommen

werden. Damit also ein Programm als Gegenstand in einer der anderen Bedeutungsdimensionen konstituiert werden kann, muss dieser in irgendeiner Form den physischen Wahrnehmungen mitgegeben, appräsentiert sein.

So sind die virtuell-physischen Gegenstände vermittelt über direkte Wahrnehmungen physischer Gegenstände konstituiert. Eine Datei lässt sich im virtuellen Raum eines Dateisystems dadurch verorten, dass dieses Dateisystem visualisiert und auf einem Monitor nachgezeichnet wird. Die Verortung in einem durch ein Kommunikationsprotokoll aufgespannten Raum (z.B. im Internet) findet durch die Wahrnehmung einer Adressierung in diesem Raum statt, z.B. durch Ausgabe einer Zahlenfolge oder durch die bildliche Darstellung eines Netzwerks. Der zeitliche Zusammenhang zwischen Versionen wird über ein Lesen der Versionsnummern hergestellt. Damit sind virtuell-physische Gegenstände ausschließlich durch Appräsentation zugänglich, eine direkte Wahrnehmung in virtuellen Räumen ist nicht möglich.

Jedes syntaktische Programm ist über physische Gegenstände appräsentiert. Syntaktische Programme sind als Folge von Zeichen darstellbar und lesbar. Um in Wahrnehmungsakten von Menschen konstituiert zu werden, muss die Zeichenfolge im Rahmen physischer Sinneseindrücke erfasst werden. Erst durch eine Interpretation dieser Eindrücke als Zeichen wird der physischen Wahrnehmung ein syntaktischer Gegenstand appräsentiert, und ein Programm kann als syntaktischer Gegenstand konstituiert werden. Im Grenzfall einer lediglich gedachten Zeichenfolge kann eine physische Verortung dieser im als physisch wahrgenommenen Selbst, „im Kopf“, konstituiert werden.

Sowohl Programme als maschinelle wie auch als natürlichsprachliche Semantik sind jeweils über die Syntax vermittelt appräsentiert. Eine Programmsemantik ist nicht wahrnehmbar, ohne dass diese in der Form von Zeichen, also syntaktisch, dargestellt wird. Das idealtypische Beispiel dafür ist die maschinelle Semantik, die einem gelesenen Quellcode zugeschrieben wird. Dabei wird dieser Quellcode in einer bestimmten Programmiersprache interpretiert und dadurch über die Sprache auf ein abstraktes Maschinenverhalten geschlossen. Analog dazu können die Schlüsselwörter, Bezeichner und Kommentare in einer natürlichen Sprache gelesen werden und darüber eine natürlichsprachliche Bedeutung des Programms konstituiert werden. Abgesehen von Quellcode sind jedoch auch formale Semantiken in einer syntaktischen Form gegeben: Wird z.B. mithilfe einer formalen Sprachspezifikation eine axiomatische Semantik aus dem Quellcode abgeleitet, um mithilfe dieser Eigenschaften des Programms zu untersuchen, so ist auch diese Semantik über eine bestimmte Syntax dargestellt. Diese syntaktische Darstellung erfolgt gerade nicht unbedingt in der verwendeten Programmiersprache, sondern meistens in einem davon abweichenden mathematischen Formalismus. Gleichwohl wird auch hier die Semantik über eine syntaktische Darstellung konstituiert, die wiederum physischen Wahrnehmungen appräsentiert ist.

Programme zur Laufzeit werden über ihre physischen Auswirkungen wahrgenommen. In vielen Fällen beschränken sich diese Auswirkungen auf Darstellungen auf einem Monitor, jedoch können auch Tonausgabe, das Verhalten eines Druckers, Elektromotoren usw. dazugehören. Oft muss die Wahrnehmung der Auswirkungen auch technisch erst hergestellt werden: Wie ein Programm einen Datenträger beschrieben hat kann nur durch die Verwendung entsprechender Lesetechnik herausgefunden werden. Welche reale Laufzeit ein Programm bei einer bestimmten Eingabe hat kann am besten mithilfe eines weiteren Programms gemessen werden. Ob ein Programm die Temperatur über eine Lüftersteuerung korrekt reguliert lässt sich mithilfe dafür passender Sensortechnik und weiteren Programmen für Aufzeichnung und Vergleich mit Vorgabewerten überprüfen.

Die maschinelle Semantik eines Programms als abstraktes Maschinenverhalten und das konkrete Maschinenverhalten eines Programms zur Laufzeit können zueinander jeweils wechselseitig appräsentiert sein. So kann die Arbeit am Quellcode, bei der eine bestimmte maschinelle Semantik erzeugt wird, sich mittelbar auf konkrete Anwendungen des Programms beziehen: Ein Programm wird geschrieben (oder überarbeitet), um etwas bestimmtes zu tun. Dieses Anwendungsszenario wird bei der Codeentwicklung aufgegriffen. Die Frage „Wie verhält sich das Programm in dieser bestimmten Situation?“ appräsentiert eine bestimmte Einbettung (die Situation) des laufenden Programms. Umgekehrt ist dem laufenden Programm die Semantik des entsprechenden Quellcodes (oder zumindest des ausgeführten Maschinencodes) appräsentiert, sobald das Maschinenverhalten als programmgesteuert wahrgenommen wird. Die maschinelle Semantik (bzw. die Eigenschaften, die durch Beobachtung des laufenden Programms über sie bekannt sind) wird dabei als (eine) Ursache des konkreten Maschinenverhaltens konstituiert. Nur durch diese Appräsentation kann zum Beispiel ein *Bug Report* stattfinden: Durch die Beobachtung eines konkreten Maschinen(-fehl-)verhaltens wird ein Fehler im appräsentierten Quellcode vermutet und dokumentiert. Durch das Mitgegebensein des syntaktischen und semantischen Programms bei der Beobachtung zur Laufzeit kann das Fehlverhalten auf diese Gegenstände bezogen werden.

Ähnlich verhält es sich mit dem Programm als strukturellem Bezugspunkt. Die Bedeutung und Funktionalität eines Programms, das in eine Organisation eingebettet ist, muss konzeptionell in die Arbeit am Quellcode einfließen: Die Anwendungsszenarien schließen hier neben anderen Programmen und den konkreten Maschinen, auf denen das Programm ausgeführt wird, menschliches Verhalten mit ein. Diese Einbettung in ein konkretes soziotechnisches System kann bei der Wahrnehmung der Semantik mit konstituiert werden. Umgekehrt wird Verhalten in diesem soziotechnischen System mit der Semantik des Computerprogramms verknüpft. Findet zum Beispiel zu einem Zeitpunkt in einer Universität in einem bestimmten Raum eine bestimmte Vorlesung statt, so liegt eine Ursache für dieses Ereignis in der Semantik der Raumplanungssoftware und eine weitere in der Einbettung der Software in die Organisation Universität. Insbesondere bei Fehlverhalten der Software kann die Appräsentation der Programmsemantik schnell deutlich expliziert werden, im Beispiel wenn zwei verschiedene Vorlesungen gleichzeitig im selben Raum stattfinden sollen und dieser Fehler moniert wird.

Auch physische Programme, die nicht direkt wahrgenommen werden, können anderen Wahrnehmungen appräsentiert sein. So kann von der physischen Wahrnehmung des Verhaltens eines Fahrkartenautomaten darauf geschlossen werden, dass dieser programmgesteuert ist. Diese indirekte Beobachtung eines Programms zur Laufzeit führt in einem weiteren Schritt dazu, dass ein physisch gespeichertes Programm konstituiert wird: Auf irgend einem Datenträger, sei es eine Festplatte, ein ROM oder lediglich flüchtiger Speicher, der mit diesem Automaten verbunden ist, muss Maschinencode zur Steuerung festgehalten sein. Dadurch ist der physischen Wahrnehmung des Automaten auch ein physisch gespeichertes Programm indirekt appräsentiert.

An diesem Beispiel kann illustriert werden, dass die Wahrnehmung physischer Programme empirisch ist und als empirische Erfahrung kein gesichertes Wissen erzeugen kann: Der Automat könnte auch durch eine komplizierte Mechanik gesteuert sein, das indirekt wahrgenommene Programm würde als solches nicht existieren. Ein auf einem Datenträger wahrgenommenes Programm kann durch einen Defekt nicht mehr lesbar sein. Die Ausgabe eines Programms kann, bevor sie auf dem Monitor sichtbar gemacht wird, durch andere Programme verfälscht werden.

Da Programme in allen Bedeutungsdimensionen physischer Gegenstände bedürfen,

auf denen sie in irgendeiner Form festgehalten sind, erstreckt sich diese Unsicherheit empirischer Erfahrung auch auf nichtphysische Programme. Ein auf Papier aufgeschriebenes Programm kann durch schlecht lesbare Handschrift oder einen Tintenfleck „falsch“ interpretiert werden („falsch“ meint hier: in einer anderen als der beim Aufschreiben intendierten Bedeutung). Darüber hinaus kann die Gegenstandskonstitution Fehler durch Überforderung des wahrnehmenden Menschen enthalten, wenn die Komplexität des erfassten Gegenstands hinreichend groß ist: Auch mit Zugriff auf den gesamten Quellcode eines Programms kann die maschinelle Semantik in den meisten Fällen nicht vollständig erfasst werden – viele Fehler in Programmen sind gerade durch diese „Überforderung“ schwer sichtbar.

Schließlich beinhaltet die Konstitution appräsentierter Gegenstände in vielen Fällen über die unsichere Wahrnehmung hinaus die Anwendung unsicherer Schlüsse. Im Beispiel des Fahrkartenautomaten ist der Schluss von seinem Verhalten auf eine Programmsteuerung bereits unsicher. In vielen Fällen soll von einem konkreten Maschinenverhalten auf eine abstrakte Semantik geschlossen werden – oder umgekehrt. Beide Schlüsse sind unsicher. Die erste Form von Schluss kommt zum Beispiel bei einer Fehlersuche vor: Das Programm produziert ein bestimmtes (so nicht gewolltes) Ergebnis, und mit dem Ziel, den Fehler zu beheben, wird eine Erklärung seines Zustandekommens konstruiert. Diese Erklärung muss anschließend am Code überprüft werden – und sie kann sich als falsch herausstellen. Die zweite Form von Schluss, von einer abstrakten Semantik auf konkretes Maschinenverhalten, ist die Prognose, wie sich ein Programm in einer konkreten Ausführung verhalten wird. Da diese Ausführung von der Einbettung in viele andere Gegenstände abhängig ist, kann die Prognose sich als falsch erweisen.

Die jeweilige Appräsentation der Bedeutungsdimensionen des Programmbegriffs ermöglicht es, unterschiedliche Gegenstände in ihrem Zusammenhang zu konstituieren. Dadurch kann im alltäglichen Umgang mit Programmen von einem Gegenstand gesprochen werden, auch wenn unter diesen Gegenstand sich stark voneinander unterscheidende Phänomene subsumiert werden. So kann von einem als Quellcode vorliegenden Programm gesprochen werden, das auf einem Datenträger gespeichert ist, das eine bestimmte Semantik hat, das auf einer konkreten Maschine läuft und das für eine bestimmte Aufgabe eingesetzt wird. Die unterschiedlichen Phänomene, die hier vereint werden, werden durch ihre wechselseitige Appräsentation als derselbe Gegenstand gedacht.

Diese Möglichkeit, Zusammenhänge zwischen Programmen in unterschiedlichen Bedeutungen des Begriffs herzustellen, ist nicht alleine aus den Gegenständen heraus erklärbar. Dass zum Beispiel einem textlichen Gegenstand, einem syntaktischen Programm, eine bestimmte Semantik zugeschrieben werden kann, liegt nicht in den Zeichen des Programms selbst begründet. Um diesen Zusammenhang herzustellen muss ein Zeichensystem, die Programmiersprache, angewendet werden, um den syntaktischen Gegenstand zu interpretieren. Damit ist die Beherrschung des Zeichensystems Voraussetzung für die Appräsentation der Semantik. Dies gilt für die zum Programmieren verwendeten formalen Sprachen genau so wie für die Verwendung natürlicher Sprache. Die Kenntnis des Zeichensystems geht also der Deutung voraus:

> „Ein *Zeichensystem* z.B. eine Sprache *beherrschen*, heißt die Zeichenbedeutung des einzelnen Zeichens innerhalb dieses Systems in expliziter Klarheit erfassen. Dies ist nur möglich, wenn das Zeichensystem und die dazugehörigen einzelnen Zeichen sowohl als Ausdrucks- als auch als Deutungsschema für vorerfahrene Akte in der Weise des Wissens präsent sind. In beiden Funktionen, als Deutungs- und als Ausdrucksschema, weist jedes Zeichen auf Erfahrungen zurück, welche seiner

> Konstituierung verangegangen sind. Als Ausdrucks- und als Deutungsschema ist ein Zeichen nur von eben jenen es konstituierenden Erlebnissen her verstehbar, die es bezeichnet; sein Sinn besteht in der Transponierbarkeit, d.h. in seiner Rückführbarkeit auf anderweitig Bekanntes. Dieses kann seinerseits entweder das Schema der Erfahrung selbst, in das das Bezeichnete eingeordnet ist, oder aber ein anderes Zeichensystem sein.“[131]

Die für die Sprachbeherrschung notwendige Vorerfahrung liegt nicht in den Zeichen des syntaktischen Programms selbst. Sie wird stattdessen in der Vermittlung und dem Erlernen der jeweils verwendeten Programmiersprache hergestellt. Sowohl bei natürlicher Sprache als auch bei Programmiersprachen kann die Vermittlung von Vorerfahrung gesellschaftlich aufwändig gestaltet sein: Von Sprachkursen, Schulunterricht, Lehrbüchern, Vorlesungen und Übungsgruppen über Prüfungen zum Nachweis der Sprachbeherrschung, Organisationen zur Normierung von Sprache und Büchern zum Festhalten dieser Normen bis zu Online-Tutorials, Programmierübungen und Lerngruppen werden zahlreiche Techniken angewendet, um die Verwendung der Zeichen als Ausdrucks- und Deutungsschemata zu harmonisieren.

Syntaktischen Programmen, also in Programmiersprachen interpretierten Zeichenketten, kommt neben der Transponierbarkeit in ein abstraktes Maschinenverhalten, in eine maschinelle Semantik, noch eine weitere Transponierbarkeit als Sinn zu: Sie sollen auch in ein konkretes Maschinenverhalten transponiert werden können. Auch die Herstellung dieser Transponierbarkeit und ihre Normierung können aufwändig sein: Sprachspezifikationen und Richtlinien für Compiler werden geschrieben und aktualisiert, Compiler, Interpreter und virtuelle Maschinen entwickelt und fortlaufend an neue Hardware angepasst, Testsuiten und Beispielprogramme bereitgestellt, Prozesse für die Entwicklung der Sprache festgelegt, Designentscheidungen in Sitzungen von Komitees und Beiräten getroffen usw. Erst durch diese Arbeit an der Sprache wird eine dauerhaft stabile Möglichkeit geschaffen, syntaktische Programme als in einer bestimmten Sprache geschrieben zu verstehen, ihnen eine Semantik zuzuschreiben und sie mit der Anwendung auf konkreten Computern zu verknüpfen.

Die Appräsentation, die den Zusammenhang zwischen den einzelnen Bedeutungsdimensionen des hier entwickelten Programmbegriffs herstellt, kann also nur als eine technisch und gesellschaftlich voraussetzungsreiche Verknüpfung von Gegenständen verstanden werden. Den Voraussetzungen kommen dabei mehrere Eigenschaften zu. Erstens können sie sowohl technischen als auch gesellschaftlichen Charakter haben. Eine Programmiersprache wird erst dadurch zu einem festen Regelwerk der Transponierung syntaktischer Programme, dass die die Sprache definierenden Normen gesellschaftlich ausgehandelt und stabilisiert werden. Gleichzeitig müssen die technischen Voraussetzungen zur Übersetzung geschaffen werden. Selbst bei einer Eigenentwicklung der Sprache muss auf technische und gesellschaftliche Voraussetzungen zurückgegriffen werden, von der Computertechnik bis zum Erlernen natürlicher Sprache und der Vermittlung von Techniken zur Softwareentwicklung.

Zweitens sind die Voraussetzungen hergestellt, und diese Herstellung kann unterschiedlich gestaltet sein und auch scheitern. So kommen beispielsweise Datenträger zum Speichern von Programmen und ihre Schreib- und Lesetechniken in einer Vielzahl unterschiedlicher Formen, Kapazitäten und Geschwindigkeiten. Auch wenn alle ihre Aufgabe erfüllen, das Programm für eine längere Zeit festzuhalten, macht es für den Zugang zum

131 Schütz, Alfred: Der sinnhafte Aufbau der sozialen Welt. Eine Einleitung in die verstehende Soziologie. UVK, 2004. S. 253f.

Programm einen großen Unterschied, ob es auf Magnetband oder USB-Stick vorliegt: Alleine die unterschiedlichen Zugriffsgeschwindigkeiten ermöglichen unterschiedliche Arbeitstechniken. Die Herstellung der Voraussetzungen zur Programmkonstitution wird besonders deutlich, wenn sie scheitert: Wenn eine Programmiersprache kaum noch verwendet wird, die Lesetechniken für alte Datenträger nicht mehr verfügbar sind oder ein Compiler fehlerhaft ist, wird die Verknüpfung von physischem Gegenstand, syntaktischer Zeichenfolge, maschineller Semantik und Anwendung zum Problem.

Dies führt zur dritten Eigenschaft, die darin besteht, dass die Voraussetzungen nicht lediglich einmal geschaffen, sondern in der Folge auch aufrechterhalten und reproduziert werden müssen. Eine offensichtliche Abhängigkeit der Computertechnik, und das schließt die zur Wahrnehmung von Programmen notwendige Technik ein, ist die von Strom. Dessen Verfügbarkeit und die gesellschaftlichen und technischen Voraussetzungen zur Verteilung und Nutzung werden mit hohem Aufwand stabilisiert und reproduziert. Weiter muss, für Computertechnik spezifischer, die Hardware hergestellt und gewartet werden. Die Software (insbesondere z.B. Compiler oder Betriebssysteme) muss fortlaufend an neue Hardware angepasst werden. Programmiersprachen müssen normiert werden (z.B. in der Festlegung einer Sprachspezifikation), und diese Normierung wird immer wieder sich ändernden Rahmenbedingungen angepasst. Auch das Erlernen und die Vermittlung von Programmiersprachen, sei es in Selbstlerneinheiten oder Universitätsvorlesungen, stellt eine solche Reproduktion der Voraussetzungen zur Programmkonstitution dar.

Viertens sind die gesellschaftlichen und technischen Voraussetzungen zur Wahrnehmung von Programmen die gleichen, die auch zur Anwendung von Programmen als Technik notwendig sind. Erst durch eine Programmiersprache, in der eine Zeichenfolge interpretiert wird, wird diese Zeichenfolge zum Programm. Gleichzeitig kann die Zeichenfolge auch nur durch diese Sprache sinnvoll bearbeitet werden oder für eine bestimmte Funktion angewendet werden. Analog dazu kann eine Festplatte erst durch die entsprechenden Schreib- und Lesetechniken als Datenträger konstituiert werden – und sie kann nur mithilfe dieser Techniken Träger eines Programms sein und als solcher eingesetzt werden.

Zusammenfassend sollen die Voraussetzungen deshalb hier als soziotechnische Bedingungen der Programmkonstitution bezeichnet werden. Diese Bedingungen lassen sich mithilfe der Akteur-Netzwerk-Theorie Bruno Latours näher analysieren. Dazu muss zunächst betont werden, dass die Bedingungen auf Programme bezogen eine vermittelnde Funktion erfüllen: Sie verknüpfen unterschiedliche Gegenstände, Programme in den unterschiedlichen hier vorgestellten Begriffsdimensionen, miteinander. Erst durch sie kann eine Assoziation zwischen physischen Computern und maschineller Semantik oder zwischen einer Zeichenfolge und der zugeschriebenen Bedeutung eines Steuerungsprogramms in einer Organisation hergestellt werden. Damit verbinden diese Bedingungen unterschiedliche Bereiche und stellen die im Programmbegriff ausgedrückte Assoziation erst her. Latour unterscheidet bei diesen „sozialen“ Bedingungen (in Latours Terminologie ist „das Soziale“ ein Verknüpfungstyp, der heterogene Teile durch Assoziation miteinander verbindet[132]) zwischen zwei Formen, in denen zwischen unterschiedlichen Bereichen vermittelt wird:

> „Ein *Zwischenglied* ist in meinem Vokabular etwas, das Bedeutung oder Kraft ohne Transformation transportiert: Mit seinem Input ist auch sein Output definiert. Für alle praktischen Belange kann ein Zwischenglied nicht nur als eine Black Box verstanden werden, sondern ebenfalls als eine Black Box, die als eine Einheit zählt, selbst wenn sie im Innern aus vielen Teilen besteht. *Mittler* andererseits

132 Latour, Bruno: Eine neue Soziologie für eine neue Gesellschaft. Einführung in die Akteur-Netzwerk-Theorie. Aus dem Englischen von Gustav Roßler. Suhrkamp, 2007. S. 17.

zählen nicht automatisch als eine Einheit; bei ihnen ist vielmehr jeweils offen, ob sie überhaupt nicht, als eine Einheit, als mehrere oder als unendlich viele zählen. Aus ihrem Input läßt sich ihr Output nie richtig vorhersagen; stets muß ihre Spezifität berücksichtigt werden. […] Mittler übersetzen, entstellen, modifizieren und transformieren die Bedeutung oder die Elemente, die sie übermitteln sollen. Ganz gleich, wie *kompliziert* ein Zwischenglied ist, für alle praktischen Belange, kann es als eines zählen – oder sogar überhaupt nicht, denn es kann leicht vergessen werden. Ganz gleich wie einfach ein Mittler aussehen mag, er kann *komplex* werden; er kann in verschiedene Richtungen führen, und jede von ihnen wird die seiner Rolle zugeschriebenen widersprüchlichen Erklärungen modifizieren. Ein ordnungsgemäß funktionierender Computer wäre ein gutes Beispiel für ein kompliziertes Zwischenglied, während ein banales Gespräch zu einer furchtbar komplizierten Kette von Mittlern werden kann, in der Einstellungen, Meinungen und Leidenschaften sich an jeder Wendung verzweigen. Doch wenn der Computer versagt, kann er sich in einen äußerst komplexen Mittler verwandeln […]“[133]

Latours Beispiel des Computers, der eben nur dadurch Zwischenglied ist, dass er ordnungsgemäß funktioniert, kann hier auf die Eigenschaft soziotechnischer Bedingungen der Programmkonstitution bezogen werden, die besagt, dass ihre Herstellung scheitern kann. Der nicht funktionierende Computer tritt nicht mehr als Zwischenglied auf. Am Beispiel wird sichtbar, dass Mittler und Zwischenglieder unter bestimmten Voraussetzungen ineinander umgewandelt werden können. Die Umwandlung eines Mittlers in ein Zwischenglied stellt dabei eine Transformation dar, die unter Aufwand hergestellt werden und aufrechterhalten werden muss, und zwar zumeist unter dem Einsatz weiterer Mittler.[134] Diese Umwandlung, hier als Stabilisierung eines Mittlers *als* Zwischenglied aufgefasst, wird sich im Folgenden als wichtiger Schritt bei der Herstellung der soziotechnischen Bedingungen der Programmkonstitution erweisen: Die im Programmbegriff enthaltenen Assoziationen lassen sich mithilfe der sie erzeugenden Mittler und deren Stabilisierung als Zwischenglieder erklären.

Wie bei der Darstellung der Appräsentationen kann hier zunächst bei den physischen Gegenständen angefangen werden. Computer, Datenträger, Speicherriegel und Papier sind zunächst greifbare Gegenstände, denen unter bestimmten Bedingungen die Funktion zukommt, ein Programm zu speichern. Um aus dem greifbaren Gegenstand ein syntaktisches Programm, eine Zeichenkette, zu erzeugen oder umgekehrt den physischen Gegenstand zum Träger dieser Zeichenkette zu machen, werden Schreib- und Lesetechniken als Mittler eingesetzt. Diese Techniken können zeitweise als Zwischenglieder stabilisiert werden – wenn die Zeichenkette geschrieben und gelesen werden kann, ohne dass es bei ihr zu Veränderungen kommt. Die Zwischenglieder können versagen – bei unleserlicher Handschrift, wenn das Laufwerk für einen Datenträger nicht mehr verfügbar ist, wenn die Lochkarten in ihrer Reihenfolge vertauscht werden usw. Die zusätzlichen Mittler, die die Stabilisierung ermöglichen, reichen dabei vom Stromnetz und Normierungen für Steckerformen über die Fabriken für Festplatten und Kabel bis zum Einzelhandel, der diese Festplatten und Kabel verkauft.

Auch die Assoziation der physischen Gegenstände mit virtuell-physischen Gegenständen bedarf der Stabilisierung von Mittlern als Zwischengliedern. So kann eine

133 Latour, Bruno: Eine neue Soziologie für eine neue Gesellschaft. Einführung in die Akteur-Netzwerk-Theorie. Aus dem Englischen von Gustav Roßler. Suhrkamp, 2007. S. 70.

134 Siehe ebda. S. 72.

Programmdatei nur so lange als Gegenstand wahrgenommen und verwendet werden, wie eine Technik zum Lesen des Dateisystems vorhanden ist. Dieser Teil des Betriebssystems fungiert hier als Zwischenglied zwischen den durch andere Techniken lesbar gemachten Bits auf dem Datenträger und der Abgrenzung eines Gegenstands als Datei. Bei zueinander inkompatiblen Dateisystemen wird sichtbar, dass die Herstellung und Aufrechterhaltung dieser Vermittlung notwendige Voraussetzung zur Verwendung von Programmen ist. Analog dazu kann eine Verortung im virtuellen Raum eines Netzwerks nur dadurch stattfinden, dass die Protokolle, auf denen dieses Netzwerk basiert, als getreue Zwischenglieder stabil gehalten werden (neben den physischen Mittlern, den Kabeln, Modems, Routern usw.).

Ein reales Beispiel, bei dem diese Stabilisierung an ihre Grenzen kommt und die Funktionalität daher unter Aufwand sichergestellt werden muss, ist IPv4 (Internet Protocol Version 4). Die Adressierung in IPv4 geschieht über eine 4 Byte lange IP-Adresse. Da die Länge von 4 Byte lediglich 2^{32} Adressen zulässt (ungefähr 4,3 Milliarden, wobei viele der Adressen reserviert sind), und jedes Gerät im Internet eine eigene Adresse haben soll, hat diese Form der Adressierung die Grenze seiner Kapazität seit einigen Jahren erreicht. Um die virtuelle Verortung weiterhin zu ermöglichen, werden neben der ihrerseits aufwändigen Umstellung auf IPv6 Techniken implementiert, die unter der Bedingung des begrenzten Adressraums die getreue Übersetzung, also die Funktionalität des Internets als Zwischenglied, aufrecht erhalten. Wird also zum Beispiel das Versionsverwaltungssystem bei einem Softwareentwicklungsprojekt über eine Internetverbindung kontaktiert, so kann nur durch diese Stabilisierung der virtuellen Verortung (sei es durch IPv6 oder IPv4 in Verbindung mit den Adresstechniken) die Ortsangabe[135] mit den Programmdateien verbunden werden.

Auch Programmiersprachen und Compiler fungieren als Mittler, die die Assoziationen zwischen syntaktischem Programm und maschineller Semantik bzw. beim Compiler zwischen verschiedenen Programmiersprachen herstellen. Im Falle ordnungsgemäßer Funktion stellen sie Zwischenglieder dar: Die Semantik wird gemäß einer sauberen Sprachspezifikation eindeutig einem syntaktischen Programm zugeschrieben, die Übersetzung zwischen zwei Sprachen verändert die Semantik nicht. Auch hier gibt es Fälle, in denen sowohl Sprache als auch Compiler zu Mittlern mit unbestimmten Output werden können: Zum Beispiel können in einer Sprache Teile der Semantik eines Programms bei der Ausführung auf mehreren Prozessoren unbestimmt bleiben, wenn unterschiedliches zeitliches Verhalten zu unterschiedlichen Ergebnissen führt. In diesen Fällen muss in der Softwareentwicklung darauf geachtet werden, das in der Sprache nicht definierte Maschinenverhalten abzufangen bzw. mit dem nicht definierten Verhalten umzugehen. In einigen Fällen können Laufzeitfehler bei Programmen, die nicht definiertes Verhalten einschließen, erst bei bestimmten Optimierungen durch den Compiler auftreten. Dann wird der Compiler für dieses Programm zum Mittler, da sein Output je nach Konfiguration Fehler im Programm überdecken kann. Weiter können sowohl Compiler als auch Sprachspezifikation Fehler enthalten – die Stabilisierung als Zwischenglied scheitert dann in bestimmten Fällen.

Teile der Softwaretechnik können als Mittler zwischen Spezifikation und Programmcode verstanden werden. So haben die verschiedenen Entwicklungstechniken, die zum Schreiben und Warten von Programmen zum Einsatz kommen, die Aufgabe, eine bestimmte Funktionalität eines Programms herzustellen, aufrechtzuerhalten und weiterzuentwickeln. Sie umfassen von Richtlinien zum Schreiben von Code über Entwurfsmuster zur Implementierung von wiederkehrenden Problemlösungen bis zu

135 In vielen Fällen findet diese Ortsangabe nicht durch eine IP-Adresse, sondern durch eine URL statt. In diesem Fall kommt ein weiterer Mittler, DNS (*Domain Name System*) zum Einsatz, der den in der URL enthaltenen Domainnamen in eine IP-Adresse übersetzt. Auch dieser Mittler ist aufwändig als Zwischenglied stabilisiert – die Zuordnung von Namen mit Adressen muss über viele Nameserver hinweg gleich gehalten werden.

Organisationstechniken für große Entwicklungsteams zahlreiche Einzeltechniken, die eine effiziente Entwicklung qualitativ hochwertiger Software ermöglichen sollen. Im Forschungsbereich der Softwaretechnik werden diese Techniken (weiter-)entwickelt, an aktuelle Rahmenbedingungen angepasst und systematisiert. Trotz systematisch strukturierter Herangehensweise handelt es sich hierbei um einen Mittler, der nicht in ein stabiles Zwischenglied umgewandelt werden kann: Das Ergebnis von Entwicklungsprojekten hängt immer auch von individueller Erfahrung, Kommunikation und weiteren Rahmenbedingungen ab, jede Softwareentwicklung ist durch Kontingenz geprägt.

Die Vorgänge bei der Softwareentwicklung lassen sich als Zusammenwirken einer Vielzahl von Mittlern zwischen unterschiedlichen Bereichen auffassen. Dies fängt bei der Anforderungsanalyse an, in der die gewünschte Funktionalität des entwickelten Programms festgelegt wird. Die Anforderungen können dabei mehr oder weniger stark formalisiert erfasst werden. Die Erhebung der Anforderungen stellt einen Mittler zwischen den Wünschen und Vorstellungen der verschiedenen mit dem Softwareprojekt verbundenen Menschen (z.B. Nutzergruppen) und Vorgaben für die Entwicklung dar. Im weiteren Entwicklungsprozess können beispielhaft der Entwurf der Softwarearchitektur, die Teamorganisation, das Projektmanagement, die Erstellung von Dokumentation, die Testentwicklung, Abnahmetests und die Fehlerbehandlung als Mittler auftreten. Keiner dieser einzelnen Teilprozesse hat ein eindeutig definiertes und unter unterschiedlichen Rahmenbedingungen reproduzierbares Ergebnis. Die systematische Behandlung dieser Aufgabenstellungen in der Softwaretechnik kann daher die Zuverlässigkeit der Mittler erhöhen, erreicht jedoch nicht die Stabilität eines Zwischenglieds, das eine gewünschte Funktionalität mit dem sie verwirklichenden Programmcode verknüpft – die Softwareentwicklung bleibt also eine anspruchsvolle Aufgabe für die Beteiligten.

Ein besonders vielseitiger Mittler, der sowohl in den zahlreichen Teilbereichen der Softwareentwicklung als auch im Programmcode selbst zum Einsatz kommt, ist natürliche Sprache. Die Bedeutung, die über natürlichsprachliche Elemente des Programmcodes transportiert wird, kann dabei gewollt und ungewollt übersetzt und modifiziert werden. Ein „TODO“-Kommentar kann je nach Kontext übersetzt werden in eine unmittelbar anzugehende Aufgabe oder in ein später anstehendes Problem, das zu entsprechender Zeit gelöst werden sollte. Diese unterschiedliche Übersetzung kann gewollt sein. Gleichzeitig gibt es jedoch auch ungewollte Bedeutungsmodifikationen – Missverständnisse. Die Namensgebung eines Bezeichners kann ungewollt auf andere Gegenstände bezogen werden, seine Aufgabe im Programm dadurch falsch aufgefasst werden. Ein Kommentar, der eine Designentscheidung begründet, kann durch fehlendes Domänenwissen gar nicht, nur zum Teil oder missverstanden werden. Natürliche Sprache ist kontingent, sie kann nicht analog zu einer formalen Sprache als stabilisiertes Zwischenglied auftreten.

An diesem Beispiel lässt sich zeigen, dass auch Mittler mit hohem Aufwand hergestellt und aufrechterhalten werden, ohne dass sie sich als Zwischenglieder stabilisieren ließen. Die Beherrschung einer natürlichen Sprache wird von Eltern und in Schulen vermittelt, die Sprache wird normiert und dadurch Schreibweisen und eine Grammatik vorgeschrieben. Wortbedeutungen werden durch ihre regelmäßige Reproduktion im Sprechen und Schreiben, aber auch durch Zuschreibungen und Erklärungen von Bedeutung in Wörterbüchern relativ stabil gehalten. Änderungen und neue Bedeutungszusammenhänge werden kommuniziert, Neologismen können in Lexika und auf Webseiten nachgeschlagen werden. Domänensprachen haben jeweils ihren eigenen Wortschatz, die Bedeutungen sind jeweils innerhalb der Domäne einigermaßen stabil. Im Ergebnis kann auch die in Programmen verwendete natürliche Sprache als Mittler fungieren, der Bedeutung jenseits der maschinellen Semantik des

Programms transportieren kann.

Bei der Ausführung eines Programms tritt das System, in das das Programm eingebettet ist, als entscheidender Mittler auf, beispielsweise der PC oder das Smartphone, auf dem es läuft. Analog zu Latours Beispiel tritt dieser Computer hierbei als kompliziertes Zwischenglied in den Hintergrund – das Programm erfüllt die intendierte Funktion, es wird korrekt in ein konkretes Maschinenverhalten übersetzt. Oder aber der Computer ist defekt und tritt dadurch als möglicherweise komplexer Mittler in den Vordergrund, die Transformation wird unvorhersehbar und es erscheinen ungewollte Modifikationen. In diesem Zusammenhang muss näher betrachtet werden, zwischen welchen Bereichen das System, in das ein laufendes Programm eingebettet ist, eigentlich vermittelt, welche Assoziationen es herstellt.

Die Voraussetzungen für den Computer als ordnungsgemäß funktionierendes Zwischenglied sind vielfältig und unter erheblichem Aufwand produziert. Zum Zeitpunkt der Laufzeit liegt das Programm in der Form von direkt vom Prozessor ausführbaren Instruktionen vor, in Maschinencode. Auch der Maschinencode hat eine maschinelle Semantik, er definiert ein abstraktes Maschinenverhalten. Als programmausführendes System übersetzt der Computer diesen Maschinencode in konkretes Maschinenverhalten, er nimmt Eingaben auf, führt Berechnungen durch, produziert Ausgaben, sei es auf dem Monitor, als Ausdruck oder in der Bewegung eines Elektromotors im Roboterarm. Die korrekte Übersetzung in Maschinenverhalten kann weit gefasst werden und sämtliche daran beteiligten technischen Geräte umfassen. In diesem Fall würde das Gesamtsystem als Zwischenglied in der Übersetzung scheitern, wenn der Elektromotor defekt ist, die Keyboardtaste klemmt oder ein Kabelbruch in der Verbindung zum Monitor vorliegt. Neben diesen Ein- und Ausgabegeräten würde auch die Funktionalität der Netzwerktechnik, die die Internetverbindung sicherstellt, und die korrekte Befestigung verwendeter Sensoren zur Stabilisierung als Zwischenglied gehören.

Eng gefasst kann die Übersetzung in konkretes Maschinenverhalten auch so aufgefasst werden, dass das ausführende System, die konkrete Maschine, sich in ihren Zuständen und den Übergängen dazwischen analog zur abstrakten Maschine der maschinellen Semantik verhält. In diesem Fall wäre die Stabilisierung des ausführenden Computers auf die korrekte Verarbeitung der maschinensprachlichen Instruktionen durch den Prozessor beschränkt. Allerdings ist auch diese enge Fassung voraussetzungsreich, technisch aufwändig stabilisiert und sie kann scheitern. So hört jede Mittlertätigkeit auf, wenn der Strom ausfällt. Die Betriebstemperatur, in der ein Prozessor gemäß seiner Spezifikation arbeitet, wird im PC über wärmetransportierende Paste, einen Kühlkörper mit großer Oberfläche und Lüfter aufrechterhalten – oft unter Rückgriff auf Sensordaten und mit einer Steuerung, die die Drehzahl der Lüfter entsprechend der gemessenen Temperaturen anpasst. Alle daran beteiligten Bauteile können Defekte aufweisen oder mit Entwurfsfehlern produziert worden sein. Dasselbe gilt für zentrale Komponenten wie den Prozessor, Arbeits- und Permanentspeicher, Speichercontroller, Leiterbahnen auf dem Motherboard, Grafikkarte usw. Ein prominentes Beispiel für einen solchen Entwurfsfehler stellt der Pentium FDIV Bug dar: Eine Reihe von Prozessormodellen wurde mit einem Fehler produziert, der bei bestimmten Fließkommadivisionen falsche Ergebnisse berechnet. Weiter ist die Ausführung eines Programms von einer korrekten Funktionalität des Betriebssystems abhängig. Ist die Speicherverwaltung defekt oder ein Treiber enthält Fehler, so kann die Vermittlung zwischen Maschinencode und Systemverhalten scheitern.

Betrachtet man den Gesamtprozess einer Programmentwicklung und der Anwendung dieses Programms auf einem System, wird deutlich, dass im gesamten Vorgang zahlreiche

Mittler und Zwischenglieder die Funktionalität von Programmen als Technik sicherstellen. Die Softwaretechnik, die Stabilisierung von Programmiersprachen, Compilerbau und Betriebssystementwicklung, der Prozessor, Peripheriegeräte usw. vermitteln zwischen den physischen, syntaktischen, semantischen und eingebetteten Gegenständen, die unter den hier vorgestellten Programmbegriff fallen. Ohne diese Vermittlungen könnten Programme keine Funktion erfüllen, sie würden ihren Charakter als Technik verlieren. Damit sind die angewendeten Mittler und Zwischenglieder die soziotechnischen Bedingungen der Programmkonstitution. Sie ermöglichen gleichzeitig sowohl die Assoziation der unterschiedlichen Bedeutungsdimensionen des Programmbegriffs als auch die Herstellung und Anwendung von Programmen, also ihre Verwendung als Technik.

Für den Programmbegriff bedeutet dies, dass zum Verständnis, was ein Computerprogramm ist, nicht lediglich eine Ansammlung heterogener Gegenstände, die unter den gleichen Begriff fallen, herangezogen werden kann. Vielmehr müssen ebendiese heterogenen Gegenstände sowohl in den für sie maßgeblichen Formen der Objektkonstitution betrachtet werden, als auch im Zusammenhang – in den Verhältnissen, in denen sie zueinander stehen. Eine unter dieser Prämisse durchgeführte Analyse von konkreten Programmen, ihren jeweiligen Bedingungen und Wirkungen, erfasst sie also sowohl in den einzelnen Bedeutungsdimensionen, die in den vorangegangenen Kapiteln vorgestellt wurden, als auch als Assoziation dieser heterogenen Gegenstände, die im Programm zusammenwirken.

Mit diesen Überlegungen ist die eigentliche Begriffsentwicklung dieser Arbeit abgeschlossen. Der herausgearbeitete Programmbegriff umfasst Programme als räumlich-zeitliche (physische und virtuell-physische) Gegenstände, Programme als syntaktische Gegenstände, Programme als (natürlichsprachlich und maschinell) semantische Gegenstände sowie Programme als eingebettete Gegenstände (zu ihrer Laufzeit und als struktureller Bezugspunkt). Gleichzeitig betont der Begriff, dass diese Bedeutungsdimensionen durch wechselseitige Appräsentation und Assoziationen miteinander verknüpft sind. Da die Assoziationen als soziotechnische Bedingungen der Programmkonstitution die Grundlage für die Herstellung und Anwendung von Computerprogrammen sind, stellt das Programm als Assoziation die fünfte Bedeutungsdimension des entwickelten Begriffs dar.

3.6 Exkurs: Das Programm als Komposition

Ein Zusammenwirken der Dinge in einem Gegenstand ist keine exklusive Eigenschaft von Computerprogrammen. Die immer wieder als Träger von Programmen genannten Datenträger, die darauf gespeicherten Dateien und natürlich auch Papier können textliche Gegenstände festhalten, Programme sind nur eine unter vielen Formen textlicher Gegenstände. Auch bei den anderen Gegenständen, die durch Text, also in irgendeiner Form geordneten Zeichen, charakterisiert sind, lassen sich Formen der Gegenstandskonstitution unterscheiden, die als Wahrnehmung auf den physischen Gegenstand, den syntaktischen Gegenstand oder den semantischen Gegenstand bezogen sind. Da sich hier Analogien zu Computerprogrammen ergeben können, sollen diese Gegenstände hier näher betrachtet werden.

Ein Paradebeispiel für solche textlichen Gegenstände, bei denen auch die Sprache auf unterschiedliche Formen der Konstitution verweist, sind Bücher. Wenn von einem Buch gesprochen wird, so kann dies zunächst den physischen Gegenstand meinen. In diesem Sinne hat ein Buch eine Form, ist zu einer bestimmten Zeit an einem bestimmten Ort, es hat physische Eigenschaften wie Gewicht, Festigkeit, Temperatur usw. Durch wiederum physische Einwirkung kann der Gegenstand verändert werden, etwa indem eine Randnotiz eingetragen oder ein Wort unterstrichen wird. In einer zweiten Bedeutung kann die Menge an Zeichen gemeint sein, die im Buch geschrieben oder gedruckt sind, analog zum syntaktischen Programm. In dieser Bedeutung wäre z.B. eine Digitalisierung durch Einscannen der einzelnen Seiten und Speicherung in einer Bilddatei ein getreues Abbild der Zeichen, während physische Eigenschaften nicht übertragen werden. Auch eine Änderung des Schriftsatzes, Textsatzes, von Farben und Seitengröße würde die Zeichen und ihre Ordnung kaum verändern[136]. Schließlich kann mit Buch auch der semantische Gegenstand, der jeweilige Inhalt gemeint sein.

Eine erste Analogie zu Programmen lässt sich in Übersetzungen ausmachen: Sowohl natürlichsprachliche Bücher als auch Programme können zwischen unterschiedlichen Sprachen übersetzt werden. In beiden Fällen kann dies durch Menschen oder automatisierte Werkzeuge, durch Programme geschehen. Nun ist bei natürlichsprachlichen Büchern die Übersetzung durch Menschen zur Zeit die Regel, eine alleinige Übersetzung durch Programme die Ausnahme, bei Computerprogrammen verhält sich dies gerade umgekehrt. Das wirft die Frage nach dem Grund für diese unterschiedliche Behandlung auf. Zum Verständnis muss auf die unterschiedlichen Bedingungen und Ziele dieser Übersetzungen eingegangen werden: Beide Übersetzungen sollen den Text syntaktisch ändern, so dass er in einer anderen Sprache vorliegt. Bei Programmen soll die Übersetzung die Semantik nicht verändern. Dies wird in einer großen Zahl rechnerisch aufwändiger aber jeweils für sich genommen anspruchsloser Einzelschritte erreicht. Bei der Übersetzung natürlicher Sprache dagegen können je nach Texttyp und Kontext völlig unterschiedliche Ziele angesetzt werden, eine gewollte Änderung der Semantik ist nicht ausgeschlossen. Bei literarischen Übersetzungen kann etwa die Abweichung von genauen Wortbedeutungen erlaubt sein, wenn dadurch andere Aspekte der übersetzten Literatur besser transportiert werden, während bei der Übersetzung einer Anleitung eines Geräts das zur Bedienung notwendige Wissen getreu wiedergegeben werden soll. Natürlichsprachliche Übersetzungen involvieren ästhetische Urteile und können dabei

136 In einer bestimmten Hinsicht wird die Anordnung der Zeichen verändert, wenn etwa durch solche Änderungen mehr oder weniger Text auf jeweils einer Seite des Buchs festgehalten wird. In diesem Fall werden Seitenzahlen verändert, die auch als Teil der Zeichenmenge aufgefasst werden können. Quellenangaben mit Seitenzahlen als Verweis auf die syntaktische Stelle, an der sich etwa ein Zitat befindet, müssen diese syntaktischen Änderungen aufgreifen.

gewollt subjektiv, von den Übersetzenden abhängig, sein. Insbesondere aber sind sie auf eine bestimmte Zielgruppe zugeschnitten: Die Übersetzung soll für eine bestimmte Gruppe von Menschen lesbar und unter ihren gegebenen Vorerfahrungen verständlich sein. In Maschinencode übersetzte Computerprogramme dagegen verlieren in der Regel ihre Eigenschaft, ohne hohen Aufwand menschenlesbar zu sein. Die natürlichsprachliche Semantik der Programme geht vollständig verloren.

Das Beispiel der übersetzten Anleitung zeigt auf, dass es auch Texte gibt, bei denen eine Analogie zu Programmen zur Laufzeit existiert: Sie sollen bestimmte Handlungen ermöglichen. Diese Handlungen können als „Ausführung“ des textlichen Gegenstands aufgefasst werden. Im bereits (in Kapitel 3.4.1) genannten Beispiel der Kochrezepte wird dies besonders deutlich: Eine Reihe von zur Ausführung notwendigen Gegenständen (die Zutaten und Kochutensilien) wird aufgelistet, die notwendigen Handlungen werden in ihrer Ausführungsreihenfolge genannt. Tatsächlich können Rezeptbücher sogar Programmierkonzepte wie Schleifen („Wiederholen Sie die Schritte *X*-*Y*, bis *Z* eintritt“) oder Unterprogramme („Bereiten Sie *X* nach Rezept auf Seite *Y* vor“) enthalten. Das Kochen oder Backen selbst lässt sich dann als eingebettete Anwendung des Rezepts begreifen. Das System, in das diese Anwendung eingebettet ist, besteht aus den notwendigen Werkzeugen (inklusive ihrer Bedingungen, wie z.B. Strom- oder Gasversorgung für Kochplatten), den Zutaten sowie den ausführenden Menschen. Ein wichtiger Unterschied zu Computerprogrammen ist hierbei, dass Rezepte den Kochvorgang zwar abstrakt beschreiben können, jedoch keine abstrakte Maschine definiert ist, die normativ das Verhalten von konkreten Systemen (von den Küchen, Utensilien, Kochenden) vorgibt[137]. In der Ausführung des Rezepts können und sollen die konkreten Umstände wie der Geschmack des Zielpublikums und die Beschaffenheit der Zutaten einbezogen werden, das Ergebnis darf subjektiv von den Kochenden abhängen. Aus diesem Grund kann Kochen als kreative Tätigkeit aufgefasst werden, während z.B. die Komprimierung einer Audiodatei nach vollständig bekanntem Algorithmus dagegen ein vorher bereits in seinen Eigenschaften festgelegtes Ergebnis produziert.

Ein weiteres Beispiel solcher textlicher Gegenstände, die bestimmte Handlungen implizieren, deren Ausführung wiederum als kreative Tätigkeit konstituiert wird, sind schriftliche Aufzeichnungen von Musik. Von einfachen Akkordangaben über Tabulaturen bis zu vollständigen mehrstimmigen Partituren werden für die Aufzeichnung musikalischer Werke Gegenstände konstituiert und angewendet, die viele Parallelen zu den hier behandelten Computerprogrammen aufweisen. Ein Vergleich dieser beiden Gegenstandsbereiche und eine Betrachtung des Computerprogramms als Komposition können aufschlussreiche Hinweise zu den Eigenschaften der als Programme konstituierten Gegenstände bieten.

Zunächst findet auch bei musikalischen Werken die Wahrnehmung über physische Sinneseindrücke statt. Produzierte Töne, aber auch greifbare Instrumente und betrachtete Notenblätter, Partituren oder Tonträger werden als physische Phänomene, die zu einer bestimmten Zeit an einem bestimmten Ort wahrnehmbar sind, konstituiert. Analog zu Programmen werden auch im Umfeld musikalischer Werke und Aufführungen Virtualisierungen von Räumen geschaffen, von der Unterteilung in Bühne und Publikumsraum in Konzertsälen, die vom konkreten physischen Raum abstrahiert, über die Unterteilung in Sitzplatzkategorien wie Loge, Rang und Parkett bis zur virtuellen Verortung von Klängen über Stereotechnik. Oktavräume auf Klaviaturen, Orgelregister und Sitzanordnungen in Orchestern sind weitere Raumstrukturierungen, die von konkreten physischen Orten abstrahiert eine Adressierung ermöglichen. Insbesondere finden aber Virtualisierungen von Zeit statt: Die in

137 Selbst Utensilien, deren Verhalten relativ strikt vorgegeben ist, können sich stark in den relevanten Eigenschaften unterscheiden. So haben unterschiedlich große Kochplatten oder die Verwendung eines Gasherds anstelle eines Induktionskochfelds relevante Auswirkungen auf die Ausführung des Rezepts.

der Musik verwendeten Unterteilungen der Zeit in Takte, Phrasen, Sätze oder Perioden, aber auch die Erfassung bestimmter Abschnitte als Strophe, Refrain, Überleitung, Solo, Exposition, Reprise usw. ist nicht physisch, sondern durch den inneren Zusammenhang des Werks gegeben. Erst in einer konkreten Aufführung wird das Zeitverhältnis dieser Einheiten zueinander als physisch konstituiert. Auch Spielzeiten, Tourneen oder der zeitliche Aufbau einzelner Aufführungen stellen der als physisch wahrgenommenen Zeit Strukturierungen gegenüber, die virtuelle zeitliche Verhältnisse ausdrücken.

Die Verschriftlichung von Musik in der Form von Noten oder Tabulaturen weist große Parallelen zu den schriftlich ausgedrückten syntaktischen Programmen auf. Beide Gegenstandsformen werden als Folge von Zeichen aus einer festgelegten Zeichenmenge konstituiert, deren Art und Anordnung jeweils eine bestimmte Semantik ausdrückt. Im Gegensatz zu einem Roman wird die Musik dabei in einer stark formalisierten Sprache ausgedrückt: Die Bedeutung einer Notenform an einer bestimmten Stelle wird z.B. als Ton einer bestimmten Tonhöhe und Dauer festgelegt. Die Tonhöhe ist dabei in der Positionierung ausgedrückt und sie hängt von Vorzeichen und verwendetem Notenschlüssel ab. Die Dauer ist jeweils relativ zu einer Taktart und einem Tempo angegeben. Die Dynamik kann zusätzlich durch eigens dafür bestimmte Zeichen angegeben werden. Bei mehrstimmigen Partituren ist in der Anordnung bzw. in Annotationen darüber hinaus ausgedrückt, durch welches Instrument der Ton erzeugt werden soll bzw. für welche Stimmlage in einem Chor er vorgesehen ist.

Die gesamte Notation ist in hohem Maße formalisiert. Die in der Sprache musikalischer Notenschrift zulässig formulierten Gegenstände sind klar festgelegt, auch wenn durch zusätzliche natürlichsprachliche Annotationen weitere Informationen vermittelt werden können. Ähnlich den Kommentaren in Programmen können diese „echt“ oder „unecht“ sein, abhängig davon, ob sie einen Unterschied für die Aufführung des annotierten Musikstücks machen. Die in Partituren ausgedrückte Semantik ist auch abstrakt: Sie abstrahiert von einer konkreten Aufführung, von konkreten Instrumenten und Musiker_innen. Ein darin festgehaltenes Violinsolo ist nicht mit einer bestimmten Violine verbunden, es kann auf jeder Violine gespielt werden. In den für Programme verwendeten Begrifflichkeiten ausgedrückt ist z.B. die Semantik einer bestimmten Partitur das Verhalten eines abstrakten Orchesters – oder eines abstrakten Chors usw. Die Semantik lässt sich auch hier ohne Rückgriff auf eine konkrete Aufführung lesen und verstehen.

In der konkreten Aufführung eines Musikstücks lassen sich weitere Parallelen zum Programm ausmachen, allerdings liegt hier auch der größte Unterschied zwischen beiden Gegenstandsbereichen. In der Aufführung ist ein Musikstück stets nur eingebettet als Gegenstand zu begreifen: Es wird von einem bestimmten Orchester, von konkreten Musiker_innen auf konkreten Instrumenten gespielt. Die Bedingungen der Aufführung stellen parallel zum Programm ein System dar, das unterschiedlich weit gefasst werden kann. Musiker_innen, Chor, Dirigent_in und Instrumente stellen je nach Stück notwendige Teile dieses Systems dar, allerdings lassen sich auch der Konzertsaal, dessen Raumakustik, Aufzeichnungsgeräte und Publikum darunter fassen. In einem erweiterten Sinne kann auch die organisatorische Einbettung als notwendiger Teil des aufführenden Systems betrachtet werden. Der entscheidende Unterschied zur Programmausführung liegt nun darin, dass die Aufführung eines Musikstücks als kreative Leistung konstituiert wird. Die Notation von Musik lässt (im Gegensatz z.B. zu einer Tonaufzeichnung) viele Freiheitsgrade darin, wie diese Musik genau gespielt wird – und die Interpretation in der Aufführung kann und soll dem aufgeführten Werk einen jeweils eigenen Charakter verleihen, der nur im Zusammenwirken von Werk und Aufführenden zustande kommt. Aus diesem Grund sind die daran beteiligten

Menschen nicht nur Teil des aufführenden Systems, sie werden vielmehr als die zentralen Akteure ausgemacht. Sowohl das ausgeführte Programm als auch das aufgeführte Musikstück sind auf die abstrakte Semantik bezogen und von einer konkreten syntaktischen Darstellung dieser Semantik abhängig – als Notenblätter bzw. Maschinencode. Die Semantik des Musikstücks lässt jedoch einen größeren Spielraum bei der Abgrenzung der Ereignisse, die als seine Aufführung konstituiert werden können.

Nurbay Irmak greift im Aufsatz *Software is an Abstract Artifact*[138] die Parallelen zwischen musikalischen Werken und Computerprogrammen auf, um eine ontologische Charakterisierung von Software vorzunehmen. Er unterscheidet dafür vier Arten von Gegenständen, *Algorithm*, *Text*, *Copy* und *Execution*, von Software. Sie entsprechen grob den hier vorgestellten Gegenstandsformen von semantischen, syntaktischen, physischen und eingebetteten Programmen. Diese erfüllen nach Irmak jeweils nicht die Voraussetzungen, um den Softwarebegriff zu füllen. Stattdessen sei ein Stück Software ein abstraktes Artefakt, vergleichbar mit einem musikalischen Werk. Diese werden nicht als platonische Ideen aufgefasst, da sie anfangen und aufhören können zu existieren und damit weder ewig noch zeitlos sind. Sowohl Software als auch musikalische Werke sind nach dieser Auffassung zwar ortlos, aber sie werden zu einer bestimmten Zeit kreiert, sie hören auf zu existieren, sobald die Autoren und alle Kopien nicht mehr existieren, sie nie wieder aufgeführt oder ausgeführt werden und keine Erinnerungen an sie existieren. Weiter kann Software nach Irmak verändert werden, ohne ihre Identität zu verlieren.

> „Both software and musical works come into existence when an author or a group of authors indicate a certain algorithm or a sound structure with the right kind of intentions; intentions to create those sorts of things. This act of creation is an act of coming up with the score or the algorithm and by writing them down creating the original copy of it. The existence of software and musical works does not solely depend on a particular copy; even if the original copy is destroyed right after its creation, the software or the musical work does not thereby cease to exist, because the author(s) might create another copy of the same artifact. The existence of these artifacts depends on their authors, copies, performances or executions and memories. The existence of software and musical works depends on the above entities, processes, events, etc. in the sense that necessarily whenever one of them exists at some time software or musical work exist at some time.“[139]

Den ontologischen Fragestellungen nach Existenz und Identität von Programmen kann an dieser Stelle nicht weiter nachgegangen werden, in der hier vorgestellten begrifflichen Fassung von Computerprogrammen steht die Erfahrung von Computerprogrammen im Vordergrund. Allerdings gibt die von Irmak analysierte Analogie von Software und musikalischen Werken den Hinweis, dass beide über die vier Gegenstandsformen *Algorithm*, *Text*, *Copy* und *Execution* hinausgehen. Software lässt sich nicht auf eine dieser Formen reduzieren, und doch ist sie von ihnen abhängig – da ihre Existenz auch von der Existenz von Kopien abhängt. Weiter ist der Akt des Kreierens sowohl beim musikalischen Komponieren als auch bei der Softwareentwicklung entscheidend für die Existenz der Werke.

Auf diese Weise intentional hergestellte musikalische Werke sind Kompositionen: Sie haben einen Aufbau, der ermöglicht, dass die einzelnen Teile zusammenwirken. Diese Teile sind einzelne Töne und die von verschiedenen Instrumenten hervorgebrachten Klänge, aber

138 Irmak, Nurbay: Software is an abstract artifact. Grazer Philosophische Studien Vol. 86 Issue 1. 2012. S. 55-72.

139 Ebda. S. 69.

auch Motive, Melodien und Themen in der Musik. Analog können Programme als Kompositionen von Schlüsselwörtern, Bezeichnern, Literalen, Operatoren usw. begriffen werden – oder aber als Komposition von Deklarationen, Methoden, Klassen, Routinen usw. So wie die musikalische Komposition mehr ist als eine lose Aneinanderreihung von Tönen kann auch das Programm als Werk verstanden werden, das durch Anordnung und wechselseitigen Bezug mehr darstellt als eine Reihe von Instruktionen. Der Begriff der Komposition betont diese Eigenschaft, da hier erst durch das Zusammenwirken ein Gegenstand als Kunstwerk konstituiert wird. Analog wird dieser Begriff auch auf Malerei bezogen verwendet, auch hier entsteht erst durch das Zusammenwirken in einer bestimmten Anordnung der gewünschte Ausdruck.

Ein weiterer wichtiger Hinweis zum gegenständlichen Charakter von Programmen ergibt sich aus der Tatsache, dass bei musikalischen Werken nicht nur das Komponieren und das schriftliche Festhalten der Ergebnisse z.B. in einer Partitur als kreativer Akt, als Schaffen von etwas Neuem, betrachtet wird. Wie oben bereits festgestellt wird auch die Aufführung, die Interpretation, von Musik als Schaffen von Neuem konstituiert. Alfred Nordmann zieht im Aufsatz *Object lessons: towards an epistemology of technoscience*[140] für die Charakterisierung von technowissenschaftlichem Wissen eine Aufführung des Pianisten Alfred Brendel heran. Das in der Aufführung (und im Beispiel für eine Schallplatte aufgenommene) neu Geschaffene geht dabei über eine werktreue Wiedergabe der Sonate Franz Schuberts hinaus. Nach Nordmann kommt in diesem neu Geschaffenen technowissenschaftliches Fertigkeitswissen zum Ausdruck: Die Fertigkeit, mit einem System umzugehen und bestimmte Phänomene zu erzeugen und zu kontrollieren, drückt sich sowohl in technowissenschaftlichen Veröffentlichungen als auch in der wahrnehmbaren Aufnahme der Schubertinterpretation Brendels aus.

> „A habit of action, achieved capability, or knowledge of control guides us even in complex situations when we can safely rely on intersubjectively available causal relations and when we are cognizant immediately of this reliability and robustness rather than merely interpreting it as grounded in and derived from general laws. The collective, reliable, publicly exhibited and intersubjectively accessible assimilation and control of how a system behaves signifies quite simply that we are able to find our way about in it. And knowing our way about is tantamount to the achievement of adapting to the circumstances of a specific, highly complex world – just as Alfred Brendel knows his way about Schubert's score and as technoscientific researchers learn to move about in the nanoworld that has been opened up to them by scanning probe microscopy."[141]

Auf Programme bezogen wird dieser Nachweis der Kontrolle über ein Systemverhalten nicht in der Ausführung eines bestehenden Programms gesehen. Dieser Akt wird nicht als kreativ aufgefasst, auch wenn der ausführende Computer erwartbare wahrnehmbare Phänomene erzeugen kann. Das *Zurechtfinden* innerhalb eines Systems – und damit technowissenschaftliches Fertigkeitswissen – drückt sich allerdings in drei verschiedenen Formen des Umgangs mit Programmen und Computern aus. Erstens kommt im Programmieren Wissen um die Kontrolle von Phänomenen innerhalb eines bestimmten Systems zum Ausdruck: Sowohl auf den Umgang mit der verwendeten Programmiersprache bezogen als auch auf den Umgang mit einem Zielsystem bezogen bringt die Entwicklung etwas Neues hervor, was jeweils über das System hinausgeht. So kann es als nennenswerte

140 Nordmann, Alfred: Object lessons: towards an epistemology of technoscience. In: Scientiae Studia Vol. 10. 2012. S. 11-31.

141 Ebda.. S. 27.

Leistung kommuniziert werden, unter den Grenzen einer gegebenen Hardware bestimmte Aufgaben mit Software zu erfüllen. Die *Demoszene* entwickelte diese Darstellung von Fertigkeitswissen zur eigenen Kunstform. Zweitens kommt solches Kontrollwissen im Umgang mit Softwarekomponenten und deren Verwendung in Programmen vor. So zeigen die Einbindung von Programmbibliotheken, die erfolgreiche Kommunikation über gegebene Protokolle oder die erfolgreiche Einbindung einer Datenbank jeweils Fertigkeitswissen über den Umgang mit diesen Systemen. Im wissenschaftlichen Bereich kann sich dies in einer neuen Optimierung, einer neuen Form der Datenintegration oder einem *Proof of Concept* einer neuen Anwendung ausdrücken. Drittens kann die Anwendung von Programmen selbst Ausdruck von erheblichem Kontrollwissen sein. So sind z.B. Einführung und Konfiguration eines Enterprise Resource Planning Systems oder die Anwendung eines Cloud Computing Systems für ein neues Aufgabenfeld Nachweis und Ausdruck des auf dieses System bezogenen Fertigkeitswissens, mit dem kontrolliert neue Phänomene erzeugt werden können.

In diesen Fällen stellt die Anwendung von Programmen selbst eine kreative Leistung dar. Das darin zum Ausdruck kommende Wissen wird als Grundlage für etwas Neues konstituiert, so wie das in der Schubertinterpretation zum Ausdruck kommende Wissen Grundlage für die Erzeugung neuer Phänomene ist. Beim Beispiel der Programmbibliotheken und bei der Integration großer Systeme kann die Anwendung der Software selbst als neues Zusammenfügen, als Komposition verstanden werden. In der vorherigen Bedeutung bezieht sich Komposition sowohl in der Musik als auch beim Programm in erster Linie auf den sprachlichen Gegenstand: Das Zusammenfügen der Teile geschieht in der Partitur oder im Quellcode, jeweils mit einer appräsentierten Semantik. Bei der Anwendung kann die Komposition das ausführende System mit einschließen, wie z.B. in der Demoszene. Die Bezeichnung als Komposition drückt dann das Zusammenwirken von System und Quellcode aus, und erst in diesem Zusammenwirken wird das kreative Werk konstituiert.

Bezieht man diese Auffassung des Zusammenwirkens in der Komposition auf den oben genannten Hinweis, dass Software mehr ist als in den Gegenstandsformen *Algorithm*, *Text*, *Copy* und *Execution* zum Ausdruck kommt, so kann das Programm noch auf einer weiteren Ebene als Komposition verstanden werden: Das Programm unter Einbezug aller vier Gegenstandsformen wird erst durch das Zusammenwirken der physischen Gegenstände mit den syntaktisch darauf festgehaltenen Bedeutungen sowie den Systemen, auf denen eine Ausführung stattfinden kann, zum eigenständigen Gegenstand. Eine solche Betrachtung konstituiert das Programm als Komposition dieser Teile. Da das Zusammenwirken von den soziotechnischen Bedingungen der Programmkonstitution ermöglicht wird, stellen sie gleichzeitig die Grundlage dieser Perspektive dar.

Diese Perspektive führt jedoch von den beobachteten Parallelen zwischen Computerprogrammen und musikalischen Werken weg zu einer Betrachtung der wechselseitigen Beziehungen, die darin zum Ausdruck kommen, dass jeweils mehrere unterschiedliche Gegenstandsformen unter die Begriffe Werk und Programm subsumiert werden. Programme, die in dieser Bedeutung als Komposition verstanden werden, sind nicht in einem kreativen Schaffensakt benennbarer Menschen komponiert worden – vielmehr wird ihre Komposition von den Assoziationen heterogener Akteure hervorgebracht. Die Komponist_innen verschwinden hinter der Unbestimmtheit über den Ursprung von Handlungen[142], die Komposition ist in dieser Bedeutung gleichzusetzen mit der Herstellung der Bedingungen, unter denen Programme als solche zu Gegenständen werden.

Kehren wir also zur vorherigen Auffassung von Programmen als Komposition zurück,

142 Vgl. Latour, Bruno: Eine neue Soziologie für eine neue Gesellschaft. Einführung in die Akteur-Netzwerk-Theorie. Aus dem Englischen von Gustav Roßler. Suhrkamp, 2007. S. 81f.

in der die Komposition auf den Quellcode bezogen das Zusammenführen der Teile zu einem (syntaktischen) Programm bezeichnet. Werden diese, vergleichbar mit musikalischen Kompositionen, den Softwareentwickler_innen als Werk zugeschrieben? Wie Max Weber bereits für die Musik festgestellt hat, ist die Bedeutung der Komponierenden erst mit der schriftlichen Notation mehrstimmiger Werke in Erscheinung getreten[143]. Durch die Verwendung von Schrift wurde die Anonymität mündlicher Überlieferungen abgelöst. Das Werk wird, z.B. durch Signatur, mit einem Menschen verknüpft, dem Autorenschaft zugeschrieben wird, der kreative Schaffensprozess wird gewürdigt. Diese Verknüpfung lässt sich ganz ähnlich bei Programmen beobachten, auch wenn der Code von mehr als einem Menschen komponiert wurde:

> „Pragmatic Programmers don't shirk from responsibility. Instead, we rejoice in accepting challenges and in making our expertise well known. If we are responsible for a design, or a piece of code, we do a job we can be proud of. [...]
>
> Craftsmen of an earlier age were proud to sign their work. You should be, too. [...]
>
> Anonymity, especially on large projects, can provide a breeding ground for sloppiness, mistakes, sloth, and bad code. It becomes too easy to see yourself as just a cog in the wheel, producing lame excuses in endless status reports instead of good code.
>
> While code must be owned, it doesn't have to be owned by an individual. In fact, Kent Beck's successful eXtreme Programming method [...] recommends communal ownership of code (but this also requires additional practices, such as pair programming, to guard against the dangers of anonymity).
>
> We want to see pride of ownership. "I wrote this, and I stand behind my work." Your signature should come to be recognized as an indicator of quality. People should see your name on a piece of code and expect it to be solid, well written, tested, and documented. A really professional job. Written by a real professional."[144]

143 Weber, Max: Die rationalen und soziologischen Grundlagen der Musik. Drei Masken Verlag, 1921. S. 67f.

144 Hunt, Andrew and Thomas, David: The Pragmatic Programmer. From Journeyman to Master. Addison-Wesley. 1999. S. 258f.

4. Anwendungen des entwickelten Begriffs

Im vorangehenden Kapitel (Kapitel 3) wurde ein Begriff für Computerprogramme entwickelt, der die unterschiedlichen Gegenstände, auf die er sich beziehen kann, in vier Bedeutungsdimensionen erfasst. In einer fünften Bedeutungsdimension, dem Programm als Assoziation, wurden die Zusammenhänge zwischen diesen unterschiedlich konstituierten Gegenständen untersucht. Durch Anwendung des entwickelten Begriffs soll nun aufgezeigt werden, wie er zu Diskursen über Technik beitragen kann.

In diesem Kapitel wird der entwickelte Programmbegriff beispielhaft auf drei Bereiche angewandt, die mit der Entwicklung von Computerprogrammen zusammenhängen. Diese Anwendungen zeigen zum einen den analytischen Mehrwert des Begriffs auf; es wird dargestellt, wie durch den Bezug zu den einzelnen Bedeutungsdimensionen und die Betrachtung ihrer Verhältnisse zueinander neue Aspekte in der Analyse von Computerprogrammen in den Fokus rücken. Zum anderen stellen die Beispiele jeweils den Einstieg in einen eigenen Gegenstandsbereich von Untersuchungen zu Programmen dar. Ausgehend von den hier vorgestellten Begriffsanwendungen ließen sich jeweils eine Reihe weitergehender Fragestellungen entwickeln, die die wechselseitigen Verhältnisse von Mensch, Technik und Gesellschaft in den Blick nehmen.

Das erste Unterkapitel beschäftigt sich mit den Tätigkeiten des Testens von Programmen und der Fehlersuche darin. Diese beiden Tätigkeitsformen stellen idealtypische Handlungsweisen in der Softwareentwicklung dar. Ein großer Teil der Entwicklungsarbeit bezieht sich auf die Überprüfung von hergestellter Software und der Behebung darin enthaltener Fehler. Auch lange nach einer ersten Veröffentlichung von Software kann die Wartung inklusive der Fehlersuche und -korrektur Aufgabe der Entwickler_innen bleiben. Als typisches Beispiel von Handlungen im Umfeld der Softwareentwicklung eröffnet diese Begriffsanwendung das gesamte Feld der Prozesse, Strukturen und Techniken, die zu diesem Zweck eingesetzt werden.

Das zweite Unterkapitel untersucht das Phänomen der formalen Verifikation von Programmen. Verifikationen sind unmittelbar auf Programme als maschinelle Semantik bezogen. Durch mathematische Methoden ermöglichen sie mathematische Beweise darüber, dass die jeweils untersuchte Semantik bestimmte für die Verifikation definierte Eigenschaften aufweist. Sie gilt als aufwändig und wird üblicherweise nur durchgeführt, wenn die korrekte Funktionalität des Programms als außerordentlich wichtig wahrgenommen wird. Diese Anwendung des Begriffs eröffnet die Bereiche der Zuschreibung gesellschaftlicher Relevanz an Programme sowie der Zuverlässigkeit und des Vertrauens in die Technik. Das Verhältnis von formalwissenschaftlicher Forschung zu realen Anwendungen wird in der Untersuchung berührt; allerdings können die Fragestellungen im Umfeld dieses Verhältnisses im Rahmen dieser Arbeit nicht in angemessenem Umfang vorgestellt werden. Insbesondere die Bezüge von Softwareentwicklung und den angewandten, technischen und praktischen Teilen der Informatik zur theoretischen Informatik und zur Mathematik bedürfen einer eigenen Untersuchung. Aus diesem Grund soll an dieser Stelle lediglich darauf hingewiesen werden, dass die damit verbundenen wissenschaftstheoretischen Überlegungen auch im Umfeld von auf Programme bezogenen Techniken von Bedeutung sind.

Das dritte Unterkapitel setzt sich mit den Phänomenen *Open Source Software* und *Freie Software* auseinander. Diese beiden Konzepte beschreiben Formen des Umgangs mit dem Quellcode von Programmen, der sich insbesondere auch in zur Veröffentlichung verwendeten

Lizenzen ausdrückt. Die beiden Konzepte überschneiden sich zu großen Teilen, jedoch nehmen sie gleichzeitig miteinander konfligierende Perspektiven auf Software ein. Die Anwendung des entwickelten Programmbegriffs auf diesen Konflikt eröffnet den Gegenstandsbereich auf Programme bezogener Normen. Dies berührt zum einen ethische Überlegungen zur Computertechnik, zum anderen rechtliche Vorgaben und Fassungen von Software, die unter anderem in der Gestaltung von Lizenzen zum Ausdruck kommen.

Abgeschlossen wird dieses Kapitel mit einem Exkurs zu Computerprogrammen und Zeit. In diesem Exkurs wird in die Vielzahl von Zeitvorstellungen, die verbunden mit Computertechnik angewendet werden, eingeführt. Dabei erweist sich eine Untersuchung mithilfe des vorgestellten Programmbegriffs als aufschlussreich, da durch diesen die jeweiligen Formen von Gegenständen, auf die bestimmte Zeitkonstruktionen bezogen sind, klar unterschieden werden können und (vermittelt über die Assoziationen) auch die Verhältnisse der Zeitkonstruktionen zueinander einer Analyse zugänglich werden.

4.1 Testing, Debugging

Softwareentwicklung besteht aus einer Reihe planvoller Tätigkeiten, deren Ziel die Verwirklichung einer bestimmten Funktionalität durch das entwickelte Programm ist. Dieses Ziel kann sehr exakt gefasst sein, z.B. als formale Spezifikation, es kann jedoch auch als grobe Vorstellung einer Anwendung oder in der Form mehrerer Anwendungsszenarien existieren, die in der Kommunikation zur Entwicklung regelmäßig neu ausgehandelt werden. Agile Softwareentwicklung greift z.B. als wichtigen Aspekt auf, dass sich Ziele und die Herangehensweisen zum Erreichen dieser im Laufe eines Entwicklungsprozesses ändern können und auf solche Änderungen flexibel reagiert werden muss, um zufriedenstellende Ergebnisse zu erreichen.

Um die Verwirklichung der Funktionalität zu überprüfen, werden mehrere Techniken angewendet, die einen Vergleich der realisierten mit der gewünschten Funktionalität ermöglichen. Diese Techniken werden hier in ihrer Gesamtheit als Programmtests verstanden. Die Bezeichnung hat in dieser Verwendung eine größere Extension als alltägliche Verwendungen, die z.B. eine statische Codeanalyse nicht als Test begreifen[145]. Insbesondere fällt auch die formale Verifikation von Programmen unter diese Bezeichnung – auch sie zielt auf das Verhältnis einer gewünschten zu einer realisierten Funktionalität ab. Da diese durch die Anwendung formaler Beweise eine relevante Besonderheit in der Technik aufweist, welche erhebliche Konsequenzen für den Umgang mit ihr und ihren Ergebnissen zur Folge hat, wird sie im nächsten Unterkapitel eingehender untersucht.

Softwaretests sind Teil des Prozesses der Verifizierung und Validierung von Software. Verifizierung bezeichnet dabei den Abgleich des vorhandenen Programms mit den gesetzten Zielen der Softwareentwicklung. Validierung hingegen beinhaltet auch die Überprüfung der (in mehr oder weniger formalisierter Form) gesetzten Ziele, sie setzt das Programm in Bezug zu den vorhandenen Wünschen und Erwartungen der Stakeholder, also z.B. der Anwender_innen oder einer Organisation, die das Programm beauftragt hat.

> „The aim of verification is to check that the software meets its stated functional and non-functional requirements. Validation, however, is a more general process. The aim of validation is to ensure that the software meets the customer's expectations. It goes beyond simply checking conformance with the specification to demonstrating that the software does what the customer expects it to do. Validation is essential because, as I discussed in Chapter 4, requirements specifications do not always reflect the real wishes or needs of system customers and users.
>
> The ultimate goal of verification and validation processes is to establish confidence that the software system is 'fit for purpose'. This means that the system must be good enough for its intended use."[146]

Der entwickelte Programmbegriff kann die Unterscheidung zwischen Verifizierung und Validierung in einer bestimmten Bedeutungsebene lokalisieren: Beide Prozesse arbeiten mit dem Quellcode, der Semantik und den physischen Systemen, auf denen das Programm

145 Eine enge Definition von Test umfasst nur solche Techniken, die hier als dynamische Tests bezeichnet werden – in diesen wird ein Teil des Programms oder das gesamte Programm ausgeführt.

146 Sommerville, Ian: Software Engineering. Ninth Edition. Pearson, 2011. S. 207.

gespeichert ist und denjenigen, auf denen es ausgeführt werden soll. Die Verifizierung fasst die Einbettung des Programms eng auf: Die ausführende Maschine inklusive Betriebssystem, weiterer Software und Peripheriegeräten wird betrachtet, die Einbettung in menschliche und organisatorische Handlungszusammenhänge jedoch ausgeblendet. Die Validierung[147] dagegen untersucht, ob das Programm als struktureller Bezugspunkt seine Funktionalität erfüllen kann – ob es im vollständigen Anwendungszusammenhang seine Aufgaben erfüllt. Damit werden die Qualitätskriterien für das Programm auf die menschlichen Bezüge zu ihm ausgeweitet. Ein Scheitern der Validierung wirkt auf die gesetzten Ziele der Softwareentwicklung zurück, etwa bei einem Akzeptanztest mit negativem Ausgang, der eine Überarbeitung der zu entwickelnden Funktionalität oder des Entwurfs der Benutzeroberfläche zur Folge hat.

Softwaretests können in statische und dynamische Tests unterteilt werden. Kriterium für diese Unterscheidung ist, dass für dynamische Tests die getestete Entität ausgeführt wird und ihr dynamisches Verhalten als Prozess untersucht werden kann. Statische Tests untersuchen das Programm nicht als eingebetteten Gegenstand – sie zielen auf die Syntax und Semantik des Quellcodes ab. Eine Folge daraus ist, dass statische Tests in der Validierung nur eine untergeordnete Rolle spielen – die Einbettung in Handlungszusammenhänge wird meistens nicht untersucht (es sei denn, die Untersuchung ist mit einer Anforderungsanalyse verknüpft). Dynamische Tests hingegen können danach klassifiziert werden, welcher Teil der Einbettung in der Ausführung genau betrachtet wird.

Statische Tests untersuchen ein Programm (oder einen Programmteil) als Quellcode, jeweils bezogen auf die über die Sprache appräsentierte Semantik. Hierbei kommen formale und nicht-formale Verfahren zum Einsatz. Für die formale Verifikation muss eine formale Spezifikation vorliegen. Das Verfahren führt dann einen Beweis darüber, dass die Programmsemantik der Spezifikation entspricht. Ein weiteres Beispiel für nach formalen Kriterien durchgeführte Analysen ist die automatisierte Untersuchung des Codes durch den Compiler. Diese findet statt, bevor übersetzter Code produziert wird, sie ist auch statisch. Sie wird unter anderem für Optimierungen eingesetzt, allerdings kann sie auch Fehler aufdecken. Dabei findet sie Programmteile, die schlicht nicht übersetzbar sind (Compilezeitfehler), z.B. da sie den syntaktischen Regeln der übersetzten Sprache nicht entsprechen. Sie findet aber auch Stellen, die zwar übersetzbar sind, allerdings mit hoher Wahrscheinlichkeit ein ungewolltes Verhalten ausdrücken. Diese werden in Form von Warnungen durch den Compiler ausgegeben, sie stellen ein wichtiges statisches Hilfsmittel zur Fehlersuche dar.

In den nicht-formalen Verfahren wird der Quellcode direkt durch Menschen untersucht. Diese Verfahren umfassen das Korrekturlesen durch die Entwickler_innen und sogenannte *Reviews*, Untersuchungen in Zusammenarbeit mit anderen Menschen. Dies können Gruppenverfahren wie *Inspections* und *Walkthroughs*[148] sein. Inspektionen gehen den Code Stück für Stück in einem standardisierten Verfahren durch, dabei wird von Beteiligten auf typische Programmierfehler und Fallstricke geachtet. Walkthroughs spielen gedanklich das Verhalten des Programms in einem bestimmten Szenario durch. Die Review-Verfahren betrachten sowohl die Syntax als auch die jeweils ausgedrückte Semantik – der Code kann sowohl nach seiner abstrakten maschinellen Bedeutung als auch z.B. nach Lesbarkeit bewertet werden. Die natürlichsprachliche Semantik ist dadurch Teil des getesteten Gegenstands. Ein Vorteil dieser Codeanalyse durch Menschen liegt darin, dass Fehler direkt im Code gesucht

147 Der Begriff *Validierung* wird nicht einheitlich verwendet. In dieser Arbeit wird unter Validierung verstanden, zu überprüfen, ob das richtige Programm entwickelt wird – ob also die spezifizierten Anforderungen den Erwartungen entsprechen. Sommerville fasst dies unter Bezug auf Barry W. Boehm als „Are we building the right product?“ zusammen. Siehe Sommerville, Ian: Software Engineering. Ninth Edition. Pearson, 2011. S. 207.

148 Auch *Walkthrough* wird nicht einheitlich verwendet. Diese Arbeit orientiert sich an der Beschreibung in Myers, Glenford J., Sandler, Corey und Badgett, Tom: The Art of Software Testing. Third Edition. Wiley, 2012. S. 34ff.

werden. Bei einem Fehlverhalten in dynamischen Tests können sie schwieriger zu lokalisieren sein:

> „Another advantage of walkthroughs, resulting in lower debugging (error-correction) costs, is the fact that when an error is found it usually is located precisely in the code as opposed to black box testing where you only receive an unexpected result. Moreover, this process frequently exposes a batch of errors, allowing the errors to be corrected later en masse. Computer-based testing, on the other hand, normally exposes only a symptom of the error (e.g., the program does not terminate or the program prints a meaningless result), and errors are usually detected and corrected one by one.
>
> These human testing methods generally are effective in finding from 30 to 70 percent of the logic-design and coding errors in typical programs. They are not effective, however, in detecting high-level design errors, such as errors made in the requirements analysis process. Note that a success rate of 30 to 70 percent doesn't mean that up to 70 percent of all errors might be found. Recall from Chapter 2 that we can never know the total number of errors in a program. Thus, what this means is that these methods are effective in finding up to 70 percent of all errors found by the end of the testing process.“[149]

Die statische Analyse betrachtet das Programm nicht als Black Box: Der Quellcode selbst wird in den Reviewprozessen untersucht. Die Beschränkung des getesteten Gegenstands auf syntaktische und semantische Programme führt dazu, dass die einzige direkt involvierte Assoziation die Verknüpfung dieser beiden Wahrnehmungsformen ist: die verwendete Programmiersprache. Allerdings fließen die anderen Begriffsdimensionen auf indirektem Weg auch in die statische Analyse ein. So sind vermittelt über die Spezifikation Anforderungen an das Programm zur Laufzeit bekannt. Wird z.B. ein bestimmtes zeitliches Verhalten erwartet und die Eigenschaften der zur Verfügung stehenden Hardware sind grob bekannt, so kann sich diese Anforderung in der Spezifikation niederschlagen, die statische Analyse kann das Programm auch auf zeitkritische Stellen hin untersuchen. Weiter fließt Wissen über die Einbettung durch die beteiligten Entwickler_innen ein, über die festgelegte Spezifikation hinaus können sogar schlecht formalisierbare Erwartungen gegenüber dem Programm im Rahmen der Analyse kommuniziert werden.

Die dynamischen Testverfahren führen Programmteile unter festgelegten Bedingungen aus und überprüfen das Verhalten des Programms in diesem Rahmen. Sie können im entwickelten Programmbegriff ausgedrückt danach unterschieden werden, welcher Teil des Systems, in das sie eingebettet sind, zum getesteten Gegenstand zählt. Eine mögliche Unterteilung der Teststufen ist dabei, Unit Tests, Integrationstests, Systemtests und Anwendungstests[150] voneinander zu unterscheiden. Die Gemeinsamkeit aller dynamischen Teststufen ist, dass sie das übersetzte Programm testen und seine Funktionalität anhand festgelegter Kriterien überprüfen. Diese Kriterien können sehr informal sein und sich über den Prozess entwickeln, z.B. wenn in einem Betatest (eine Form von Anwendungstest) Probleme mit der bisher angestrebten Funktionalität aufgedeckt werden. Hier wird auch die Validierung

149 Myers, Glenford J., Sandler, Corey und Badgett, Tom: The Art of Software Testing. Third Edition. Wiley, 2012. S. 21.

150 Für einen umfassenden Überblick über die verschiedenen Formen von Softwaretests und ihre jeweilige Gestaltung siehe Sommerville, Ian: Software Engineering. Ninth Edition. Pearson, 2011. S. 210ff.

Sommerville grenzt die verschiedenen Teststufen anders voneinander ab, als es in dieser Arbeit vorgenommen wird. In seiner Fassung wird zwischen Development Testing, Release Testing und User Testing unterschieden, wobei sich Development Testing aus Unit Tests, Component Tests und System Tests zusammensetzt.

berührt, der genannte Betatest könnte z.B. zu einer Überarbeitung der Spezifikation führen.

Unit Tests überprüfen einen einzelnen, abgrenzbaren Teil des Gesamtprogramms. Diese Einheit kann ein Unterprogramm, eine Funktion, eine Klasse usw. sein (je nach verwendetem Programmierparadigma). Die Komponente wird hier losgelöst von jeder Interaktion mit anderen Programmteilen untersucht. Eingaben in den Programmteil, z.B. übergebene Werte oder Variablen, werden speziell zum jeweiligen Testfall erzeugt. Der Programmteil wird als eingebetteter Gegenstand untersucht; sowohl der Compiler als auch ein System, auf dem der getestete Programmteil läuft, sind involviert. Allerdings werden von diesem System alle Teile ausgeblendet oder simuliert, die zum Testzweck ersetzt werden können. Im idealtypischen Fall sind nach der Übersetzung der getesteten Komponente lediglich Prozessor, Speicher, Betriebssystem und die zum Testzweck erzeugten Eingaben in die Komponente beteiligt. Alle anderen Teile des Laufzeitsystems werden ignoriert, z.B. findet auch keine Ein- und Ausgabe über Peripheriegeräte statt. Unit Tests können üblicherweise automatisiert erfolgen. Jeder dafür definierte Testfall erzeugt dann eine spezifische Eingabe für die Komponente und überprüft, ob das Verhalten der Komponente den Anforderungen entspricht. Die Semantik wird insofern nur indirekt getestet, dass lediglich das eingebettete Verhalten nach Übersetzung durch einen gegebenen Compiler überprüft wird. Ein fehlgeschlagener Test kann durch eine fehlerhafte Semantik erfolgen – jedoch auch durch einen fehlerhaften Compiler oder einen Hardwarefehler im Prozessor. Unter der Annahme der Zuverlässigkeit dieser Systeme geben Unit Tests frühzeitig Hinweise auf eine fehlerhafte Implementierung einzelner Komponenten.

Integrationstests untersuchen das Zusammenwirken mehrerer Komponenten. Hierfür werden die Komponenten zusammen übersetzt und mit Eingaben versehen, die eine Interaktion zwischen ihnen zur Folge haben. Die Anzahl der Komponenten, die jeweils untersucht werden, kann sich stark unterscheiden und geht vom Test des Interfaces zwischen zwei Komponenten, die jeweils einzeln durch Unit Tests untersucht wurden, bis zur Integration großer Teile des Gesamtprogramms und der Untersuchung des gemeinsamen Verhaltens. Mit der Zahl der integrierten Komponenten steigt die Anzahl möglicher Interaktionsverläufe stark an, daher können Integrationstests auf unterschiedlichen Ebenen mit unterschiedlichen Teilmengen des Gesamtprogramms erfolgen. Auch sie laufen meist automatisiert ab; auch bei Integrationstests wird das ausführende System bis auf seine zentralen Komponenten ausgeblendet. Allerdings werden die untersuchten Komponenten als in einem Kontext agierend aufgefasst, ihr Zusammenwirken steht im Vordergrund.

Die genaue Charakterisierung von Systemtests wird unterschiedlich aufgefasst. Ian Sommerville hebt hervor, dass bei Systemtests die Komponenten von unterschiedlichen Entwickler_innen oder Gruppen zusammengefügt werden und auch das Zusammenwirken mit externen Komponenten überprüft wird[151]. In diesem Fall ist die Abgrenzung zu Integrationstests (bei Sommerville Komponententests) lediglich durch die Frage nach Urheberschaft der Komponenten festgelegt. Glenford J. Myers et al. dagegen fassen Systemtests als Vergleich des Programms mit den ursprünglichen Zielen der Entwicklung auf[152]. In diesem Fall schließen Systemtests die Interaktion mit Anwender_innen ein, z.B. sind Usability Tests ein Teil der Systemtests. Anhand des entwickelten Programmbegriffs kann hier eine andere Abgrenzung vorgeschlagen werden: Systemtests sind diejenigen Tests, die das gesamte Laufzeitsystem involvieren, in das das Programm eingebettet ist. Dies umfasst das Verhalten externer Hardware, die Bandbreite unterschiedlicher Laufzeitsystemtypen und die Interaktion mit anderer Software. Es umfasst jedoch nicht die Funktionalität als struktureller

151 Siehe Sommerville, Ian: Software Engineering. Ninth Edition. Pearson, 2011. S. 219.

152 Myers, Glenford J., Sandler, Corey und Badgett, Tom: The Art of Software Testing. Third Edition. Wiley, 2012. S. 119f.

Bezugspunkt – und damit Fragen der Benutzbarkeit, der Einbindung in Organisationen oder Akzeptanztests.

Diese fallen dann unter Anwendungstests, die die Einbettung des Programms breit auffassen und alle Stakeholder mit einbeziehen. Insbesondere fallen viele Fragen der Validität entwickelter Software unter diese Tests – allerdings nicht alle, da die Validität auch Anforderungen an Interoperabilität beinhalten, die von Systemtests abgedeckt werden. Anwendungstests würden in dieser Fassung alle Softwaretests umfassen, die Anwender_innen mit einschließen, so z.B. Alpha-, Beta- und Akzeptanztests. Alphatests untersuchen die Anwendung von Programmen in kontrolliertem Umfeld, insbesondere kann das Laufzeitsystem beschränkt sein. Betatests finden direkt im Anwendungszusammenhang statt, sie erfolgen in erster Linie durch die Anwender_innen. Akzeptanztests schließlich erfolgen durch Auftraggeber oder Kunden, die mithilfe der Tests entscheiden, ob das System akzeptiert wird und eine Abnahme stattfindet[153]. Unabhängig davon, wie die genaue Abgrenzung zwischen System- und Anwendungstests definiert wird, zeigt sich, dass die Unterscheidung verschiedener Klassen dynamischer Tests in erster Linie über die jeweilige Abgrenzung des Zielsystems stattfindet, in das das Programm eingebettet betrachtet wird.

Wird bei einem dieser Tests nun eine Abweichung zwischen den gewünschten Eigenschaften und den erhobenen Eigenschaften des Programms festgestellt, so wird dieser als Hinweis auf einen oder mehrere Fehler im Programm aufgefasst. Bei streng formulierten Akzeptanzkriterien für die Tests kann sich dieser Hinweis als eindeutige Verfehlung der Spezifikation darstellen, beispielsweise wenn eine getestete Komponente bei einer bestimmten Eingabe eine Ausgabe liefert, die nicht im für den Input definierten Bereich liegt. Bei weicher formulierten Kriterien können auch die Hinweise weniger eindeutig sein, etwa wenn die Menügestaltung nicht von allen Teilnehmenden eines Usability Tests als intuitiv wahrgenommen wurde.

In Fällen, in denen das Programm als nicht valide identifiziert wird, also sich nicht als seinem Anwendungsgebiet entsprechend entworfen darstellt, können solche Hinweise zu Änderungen der Spezifikation führen. In Fällen, in denen das Programm nicht der Spezifikation entspricht, kann der fehlgeschlagene Test Prozesse zur Fehlersuche und Fehlerkorrektur zur Folge haben. Dies muss nicht unmittelbar geschehen, Fehler können nach Relevanzkriterien klassifiziert und eine Behebung als mehr oder weniger dringlich angesehen werden. Die Prozesse der Fehlersuche und -korrektur sind dahingehend interessant, dass in ihrem Verlauf das Programm in unterschiedlichen Bedeutungsdimensionen konstituiert wird und erst durch den Zusammenhang dieser verschiedenen Gegenstände solche Debugging-Prozesse erfolgreich sein können.

Zunächst ist an dieser Stelle hervorzuheben, dass auch beim Testen und Debuggen jede direkte Wahrnehmung von Programmen über als physisch intendierte Sinneseindrücke zustande kommt. Die Werkzeuge zum automatisierten Testen etwa müssen Informationen über Gelingen und Misslingen in zugänglicher Form erfahrbar machen. Typische Werkzeuge zeigen die (Nicht-) Akzeptanz einzelner Tests etwa in Farbkodierungen an. Weiter müssen auch die Auslöser für eine Nichtakzeptanz erfahrbar gemacht werden – wenn etwa eine bestimmte Ausgabe nicht akzeptiert wurde, so kann der genaue Inhalt der Ausgabe für die Fehlersuche wichtig sein.

Im Bezug auf statische Codeanalysen wurde bereits festgestellt, dass das Programm als eingebetteter Gegenstand ausgeblendet wird und nur über die Spezifikation und das Wissen der beteiligten Entwickler_innen in den Prozess eingebracht wird. Die Fehlersuche ist durch

153 Siehe Sommerville, Ian: Software Engineering. Ninth Edition. Pearson, 2011. S. 228ff.

die direkte Lokalisierung von Fehlern bereits in der Analyse verortet. Allerdings kann die Korrektur trotzdem aufwändig sein, etwa wenn konzeptionelle Fehler involviert sind oder eine Korrektur Änderungen in völlig anderen Programmteilen voraussetzt. Die Codeanalyse bezieht sich auf das syntaktische und semantische Programm, die Einbettung ist lediglich appräsentiert. Allerdings ist diese Appräsentation von großer Relevanz: Durch sie können Laufzeitfehler entdeckt und behoben werden, ohne dass das Programm läuft.

Im Fall dynamischer Tests und einer Fehlerkorrektur bei Nichtakzeptanz sind alle vier Bedeutungsdimensionen sowie ihre Zusammenhänge involviert. Das Testwerkzeug oder der manuell ausgeführte Test machen ein Fehlverhalten physisch erfahrbar – als Beispiel kann hier der wahrnehmbare *Bugreport* eines Betatesters herangezogen werden. Dieser beschreibt mehr oder weniger genau, unter welchen Umständen der Ausführung welcher Fehler aufgetreten ist. Der Fehler wird in diesem Fall als Laufzeitfehler verstanden, der unter den beschriebenen Bedingungen zustande gekommen ist. Dieses Verhalten des eingebetteten Gegenstands wird nun herangezogen, um im wiederum physisch erfahrbar gemachten syntaktischen Programm nach dem Fehler zu suchen. Diese Suche muss dabei aufgreifen, welche Eigenschaften der Programmsemantik das beschriebene Programmverhalten verursachen kann. Die Fehlerkorrektur wiederum verändert vermittelt über das syntaktische Programm diese Semantik, um schließlich ein anderes Laufzeitverhalten zu erreichen.

Dieses Beispiel zeigt, dass in einer solchen Fehlerkorrektur auch alle Assoziationen des Programmbegriffs involviert sind. Die Techniken zum Lesen und Schreiben der physischen Gegenstände, auf denen das Programm gespeichert ist, sowie die Programmiersprache, der Compiler und das Laufzeitsystem in seiner gesamten Komplexität fließen in den Prozess mit ein. Erst durch die Zuverlässigkeit dieser Komponenten, durch die Stabilität der soziotechnischen Bedingungen der Programmkonstitution, können diese vielfachen Vermittlungen, die im Fehlerkorrekturprozess durchlaufen werden, problemlos stattfinden und das Programmverhalten effektiv angepasst werden. Die Vermittlung durch Schreib- und Lesetechniken, Compiler, Programmiersprachen usw. wird durch ihre jeweilige Stabilisierung als Zwischenglied transparent und kann ausgeblendet werden.

4.2 Formale Verifikation

Die formale Verifikation von Programmen ist eine besondere Form statischer Codeanalyse. Wie bei den anderen Analyseverfahren wird hier der Quellcode untersucht, ohne auf eine Übersetzung in Maschinensprache oder eine Ausführung des Programms zurückzugreifen. Stattdessen zielt die formale Verifikation auf die im syntaktischen Programm ausgedrückte Semantik ab: In ihr werden formale Beweisverfahren angewendet, um bestimmte Eigenschaften der Semantik sicherzustellen. Die überprüften Eigenschaften sind dabei in den festgelegten Anforderungen an das Programm begründet. Als Beweis der vollständigen (bzw. totalen) Korrektheit stellt die Verifikation einen formalen Beweis darüber dar, dass das untersuchte Programm einer formalen Spezifikation entspricht. Je nach verwendetem Verfahren kann ein solcher Beweis vollständiger Korrektheit zusammengesetzt sein aus einem Beweis partieller Korrektheit (nach Ausführung und Terminierung des Programms liegt ein den Spezifikationen entsprechendes Ergebnis vor) und einem Beweis der Terminierung des Programms. *Terminierung* und *Ausführung* sind hierbei auf die abstrakte Maschine bezogen, deren Verhalten die Semantik beschreibt.

Als statische Codeanalyse ist auch die formale Verifikation Teil des *V & V*-Prozesses, der Verifizierung und Validierung von Programmen. Allerdings gibt sie, analog zu anderen statischen Verfahren, in den meisten Fällen keine Hinweise zur Validität von Programmen. Eine strikte Festlegung der gewünschten Programmeigenschaften in einer formalen Spezifikation ist eine maßgebliche Voraussetzung für die Anwendung der Beweisverfahren. Formale Verfahren zur Validierung, beispielsweise mathematische Beweise bestimmter Spezifikationseigenschaften wie Konsistenz, fallen nicht unter die formale Verifikation. Allerdings können diese Verfahren in konkreten Entwicklungsprojekten sinnvollerweise kombiniert werden, insbesondere weil durch formale Verifikationsverfahren der Korrektheit der Spezifikation eine noch größere Bedeutung zukommt. Liegt eine formale Spezifikation vor kann durch die formale Verifikation nachgewiesen werden, dass ein Programm dieser entspricht. Allerdings kann auch versucht werden, direkt aus der Spezifikation ein Programm abzuleiten:

> „Formal methods may be used at different stages in the V & V process:
>
> 1. A formal specification of the system may be developed and mathematically analyzed for inconsistency. This technique is effective in discovering specification errors and omissions. Model checking, discussed in the next section, is one approach to specification analysis.
>
> 2. You can formally verify, using mathematical arguments, that the code of a software system is consistent with its specification. This requires a formal specification. It is effective in discovering programming and some design errors.
>
> Because of the wide semantic gap between a formal system specification and program code, it is difficult to prove that a separately developed program is consistent with its specification. Work on program verification is now, therefore, based on transformational development. In a transformational development process, a formal specification is transformed through a series of representations to program code. Software tools support the development of the transformations

and help verify that corresponding representations of the system are consistent."[154]

Auch eine solche direkte Überführung der Spezifikation in Programme (je nach genauer Gestaltung auch *program derivation* oder *transformational programming* genannt) stellt über formale Methoden sicher, dass das Programm der formalen Spezifikation entspricht. Gegenüber den anderen in der Verifizierung und Validierung angewendeten Verfahren wie z.B. dynamischen Testverfahren weist die formale Verifikation eine Besonderheit auf: Durch den Einsatz formaler Methoden kann der Begriff Verifikation speziell in diesem Verfahren streng verstanden werden, formale Verifikation ist ein *Beweis* der Wahrheit bestimmter Aussagen. Diese Beweise können geführt werden, weil sowohl die Spezifikation als auch die maschinelle Semantik des Programms jeweils formalisiert sind und damit formalwissenschaftlichen Methoden zugänglich sind. Das Programm zur Laufzeit dagegen ist in physische Gegenstände eingebettet, es läuft auf einem physischen Prozessor, verwendet physischen Arbeitsspeicher usw. Auch automatisierte Unit Tests greifen auf eine physische Ausführung des Programms auf dem Prozessor zurück. Durch diese Einbettung in als physisch erfahrene Gegenstände sind die aus den dynamischen Tests gewonnenen Erkenntnisse damit empirisch – während die Beweise einer formalen Verifikation analytisch sind.

Formale Verifikation und dynamische Tests beziehen sich also auf unterschiedliche Gegenstände. In der hier vorgestellten Terminologie zielen sie auf unterschiedliche Bedeutungsdimensionen des Programmbegriffs ab – die formale Verifikation beweist Eigenschaften der über ein syntaktisches Programm gegebenen Semantik, während dynamische Tests das aus dem syntaktischen Programm generierte eingebettete Programm zur Laufzeit untersuchen. Dem empirischen Charakter dieser Untersuchung geschuldet können dynamische Testverfahren lediglich Hypothesen bezüglich der Eigenschaften eines Programms falsifizieren – während die formalen Verfahren zum Beweis der Entsprechung von Semantik und formaler Spezifikation eine Verifikation der Übereinstimmung darstellen.

Allerdings können dynamische Testverfahren darauf hinweisen, dass Fehler existieren. Dabei werden diese Fehler im Allgemeinen nicht verortet: Sie können in der Spezifikation liegen (z.B. bei Inkonsistenzen in derselben), sie können im Verhältnis von Quellcode zu Spezifikation liegen (dies wäre ein klassischer Softwarefehler), sie können in der Übersetzung, im Compiler liegen, sie können in allen beteiligten Teilen des Systems, in das das laufende Programm eingebettet ist, liegen. So kann ein dynamischer Test auch aufgrund von Fehlern im Betriebssystem, Fehlern in der Hardware oder Anwendungsfehlern (z.B. durch Betatester) fehlschlagen. Weiter können Fehler in den dynamischen Tests selbst liegen, z.B. in der Operationalisierung der Spezifikation, in den generierten Eingaben, in der Auswertung von Ausgaben usw. Formale Verifikation dagegen kann viele Fehler gar nicht nachweisen, sie kann ausschließlich beweisen oder widerlegen, dass Semantik und Spezifikation übereinstimmen. Eine gescheiterte Verifikation, die die Übereinstimmung weder beweisen noch widerlegen kann, sagt nichts über die An- oder Abwesenheit von Fehlern aus. Und sie sagt nichts über die Validität der formalen Spezifikation aus.

Diese Eigenschaften der jeweiligen auf Programme bezogenen Techniken weisen darauf hin, dass in dynamischen Testverfahren und formalen Verifikationen unterschiedliche Formen von Wissen generiert werden. Formale Verifikation entspricht dabei dem Erkenntnisfortschritt in den Formalwissenschaften: Von festgelegten Axiomen ausgehend werden Eigenschaften abstrakter Objekte bewiesen. Diese abstrakten Objekte, die formalen Semantiken von Programmen, können dabei als mathematische Modelle des syntaktisch

154 Sommerville, Ian: Software Engineering. Ninth Edition. Pearson, 2011. S. 396.

vorliegenden Quellcodes verstanden werden. Hierbei ist relevant, dass es sich um eine Modellbeziehung handelt: Das syntaktische Programm, die Folge von Zeichen, die als auf einem konkreten physischen Objekt festgehalten konstituiert wird, ist nicht mit der über die Programmiersprache und eine dafür festgelegte formale Sprachsemantik generierten mathematischen Modellierung, einer formalen Programmsemantik, gleichzusetzen. Diese Unterscheidung bleibt auch bestehen, wenn z.B. weitere Teile eines Laufzeitsystems mathematisch modelliert werden:

> „It is of course true that mathematics allows us, amongst many other things, to model existing software/hardware systems so that we can better understand what we have built. But it is plain wrong to believe the following popular statement made in computer science:
>
> > We can investigate a mathematical model of the system to *mathematically prove* that the system will always satisfy certain requirements.
>
> We can prove that a model will satisfy certain requirements but not that the system will. If we find mistakes in our mathematical model, then this does *not neccessarily* imply that the computer program under scrutiny (or, to give another example, a digital circuit) contains flaws. Likewise, if we show that something cannot be done according to a model, then this does *not exclude* the possibility that the engineered counterpart can be done in practice after all.
>
> According to the philosophy of technology, claiming that some mathematical model *is* the real system is similar to claiming that the number 5 *is* my left hand. While my left hand can be modeled with the number 5 it cannot be replaced by it. Likewise, my laptop can be modeled with some mathematical machine model of computation, such as the universal Turing machine, but it cannot be replaced by it.“[155]

Die formale Verifikation bedient sich formalwissenschaftlicher Methoden, um gesichertes (bewiesenes) Wissen über die abstrakten Objekte, die formalen Programmsemantiken, zu erlangen. Dynamische Testverfahren dagegen, von Unit Tests bis hin zu Betatests, generieren eine andere Form von Wissen über eine andere Kategorie von Gegenständen. Die Gegenstände sind hierbei ausgeführte Programme, die in Laufzeitsysteme eingebettet sind. Ihre Erkenntnisproduktion entspricht derjenigen in den empirischen Wissenschaften: Getestet werden Hypothesen über das Verhalten eines Programms bei seiner Ausführung. Diese Hypothesen können die Produktion korrekter Ergebnisse betreffen („Das Programm liefert für jede Eingabe *X* eine Ausgabe im Bereich *f(X)*“), sie können aber auch Wechselwirkungen mit anderen Teilen des Laufzeitsystems („Funktion *X* liefert auf allen Zielsystemen, die den Mindestanforderungen der Spezifikation entsprechen, innerhalb von 200ms ein Ergebnis“) oder menschliches Verhalten („Durchschnittliche Nutzer benötigen nicht mehr als 10 Minuten zur Konfiguration“) einschließen. Diese Hypothesen werden für den jeweiligen Test operationalisiert, sein Ablauf entspricht einem experimentellen Verfahren zur Wissensgenerierung. Der Aufbau dieser Experimente kann dabei von präzisen technischen Messverfahren (z.B. Stresstests und Performance Tests, aber auch Unit Tests) bis hin zu sozialwissenschaftlichen Forschungsmethoden in Usability Tests sehr unterschiedlich gestaltet sein. Im Gegensatz zur formalen Verifikation können einige der verschiedenen Testverfahren sowohl für die Verifizierung als auch für die Validierung des Programms eingesetzt werden.

155 Daylight, Edgar G.: Turing Tales. Lonely Scholar, 2016. S. 144.

Die Sicherheit der mathematischen Beweise gegenüber den hypothesentestenden Verfahren in Experimenten legen nun die Frage nahe, warum die Übereinstimmung von Programm und Spezifikation im Regelfall nicht formal bewiesen, sondern ausschließlich durch bestätigende Tests unterlegt wird. Zunächst sind die Voraussetzungen für eine formale Verifikation in vielen Fällen nicht gegeben: Eine formale Spezifikation zu erstellen ist aufwändig und bezüglich der Validität fehleranfällig. In vielen Entwicklungsprojekten entwickelt sich die Spezifikation über die Projektlaufzeit weiter, beispielsweise wenn über die Rückmeldungen in agilen Verfahren neue Anforderungen bekannt werden. Selbst wenn die Spezifikation am Anfang eines Entwicklungsprojekts formal definiert werden kann und sich in dessen Verlauf nicht ändert, ist der Entwurf derselben auf den Aufwand bezogen mit der Programmentwicklung vergleichbar und der Prozess ist fehleranfällig. Weiter ist die formale Verifikation extrem aufwändig, bei komplexen Programmen kann eine vollständige Verifikation der Semantik nicht in realistischer Zeit durchführbar sein. Auch einer Automatisierung sind Grenzen gesetzt: Im Allgemeinen ist eine formale Verifikation nicht entscheidbar[156].

Im Ergebnis werden formale Verifikationsverfahren insbesondere für sicherheitskritische Programme oder sicherheitskritische Programmteile eingesetzt. Auf diese Weise wird der Aufwand nur dann geleistet, wenn er durch die Folgen von Programmfehlern im verifizierten Code gerechtfertigt werden kann. Da die Validität von der Verifikation nicht berührt wird, können die formalen Verfahren auch mit dynamischen Testverfahren kombiniert werden – auf diese Weise soll neben dem Beweis, dass das Programm richtig entwickelt wurde, auch sichergestellt werden, dass das richtige Programm entwickelt wurde[157]. In ihrer Gesamtheit sollen die Verfahren der Verifizierung und Validierung das Vertrauen in das entwickelte Programm erhöhen: Sie sollen zeigen, dass es die Anforderungen erfüllt, den Wünschen und Erwartungen entspricht. Durch sie soll sichergestellt werden, dass das Programm die Aufgaben, die ihm in der Anwendung zukommen, löst, dass es ihnen angemessen ist.

Die Frage, wie diese Verfahren nun genau zu einem höheren Vertrauen in die Anwendung des entwickelten Programms führen, stellt sich unter Betrachtung mithilfe des hier entwickelten Begriffs als komplex dar. Bei der Anwendung dynamischer Testverfahren sagen die in den Tests nicht falsifizierten Hypothesen jeweils aus: Ein aus dem gleichen Quellcode generiertes Programm war eingebettet in einen anderen Kontext einer möglicherweise gleichartigen Aufgabe angemessen. Unit Tests und Integrationstests sind weiter von der Anwendung entfernt: Ein aus einem Teil des Quellcodes generiertes Programm hat eingebettet in einen anderen Kontext einer Teilaufgabe, die möglicherweise in dieser Anwendung eine Rolle spielt, der Spezifikation entsprochen. An diesen Formulierungen werden mehrere Eigenschaften des Verhältnisses von Test und Anwendung deutlich. Erstens müssen für eine Bestätigung durch Tests die in der Anwendung gestellten Aufgaben mit den Aufgaben in den Tests Zusammenhänge oder Gemeinsamkeiten aufweisen. Unit Tests für Programmkomponenten, die in der Anwendung nicht ausgeführt werden, sagen nichts über diese spezielle Anwendung aus. Ein Systemtest für eine bestimmte Aufgabe sagt nichts über die Eignung des Programms für andere Aufgaben aus. Zweitens spielen Parallelen zwischen den Systemen, in die Test und Anwendung eingebettet sind, eine Rolle. Ein Test auf einem System sagt nur bedingt etwas über die korrekte Ausführung des Programms auf einem anderen System aus. Die Bedingungen hierfür reichen von Kompatibilitäten der Betriebssysteme über die Verwendung vergleichbarer Eingabegeräte bis hin zu vergleichbaren

156 Siehe z.B. Sipser, Michael: Introduction to the Theory of Computation. Third Edition. Cengage Learning, 2012. S. 201ff.

157 Vgl. Sommerville, Ian: Software Engineering. Ninth Edition. Pearson, 2011. S. 207.

Wissensbeständen der Anwender_innen. Ein Usability Test mit Schulungen im Programm sagt nichts über die Benutzbarkeit durch ungeschulte Anwender_innen aus. Insbesondere aber sind Test und Anwendung nur dann in ein Verhältnis zu bringen, wenn der jeweils beteiligte Compiler für beide Systeme korrekt funktioniert und die Systeme zuverlässig die Eigenschaften jeweils der abstrakten Maschine abbilden, deren Verhalten das übersetzte Programm beschreibt.

Formale Verifikation dagegen soll auf einem anderen Weg zu erhöhtem Vertrauen in die Funktionalität des Programms führen. Bei ihr wird aus dem Quellcode ein mathematisches Modell für ihn erzeugt – eine formale Semantik. Die Aussage des Verifikationsbeweises ist: Es ist sicher, dass eine aus dem gleichen Quellcode generierte formale Semantik die Anforderungen der formalen Spezifikation erfüllt. Ein möglicherweise erhöhtes Vertrauen in das verifizierte Programm steht hier im Zusammenhang mit der Programmiersprache (durch sie wird die Semantik erzeugt), der Zuverlässigkeit des Compilers (der Maschinencode mit gleicher Semantik erzeugen soll) und der Zuverlässigkeit des Systems, auf dem das Programm ausgeführt wird. Weiter ist sie abhängig davon, dass die formale Spezifikation für die tatsächliche Anwendungsaufgabe angemessen ist. Die Anwendung von Softwaretests und insbesondere die Anwendung formaler Verifikationsverfahren zeigen hier beispielhaft, wie Vertrauen über viele Mittler und Zwischenglieder hinweg transportiert werden kann: Der Beweis, dass ein Gegenstand (die Programmsemantik) bestimmte Eigenschaften hat (sie erfüllt die formale Spezifikation), erhöht das Vertrauen darin, dass ein kategorial anderer Gegenstand (das eingebettete Programm) andere Eigenschaften (Zuverlässigkeit der Funktionalität, Eignung für Aufgabe) aufweist. Die Stabilität dieser Vermittlungen schlägt sich darin nieder, dass Verfahren der Verifizierung und Validierung in praktisch jedem Softwareentwicklungsprojekt eingesetzt werden.

4.3 Open Source und Freie Software

Computerprogramme stehen in vielfältigen Zusammenhängen mit ethischen Überlegungen, gesellschaftlichen Normen und menschlichen Handlungen, die sich in irgendeiner Form auf Ethik oder Normen beziehen. Jeder dieser Bereiche eröffnet ein eigenes Feld heterogener wissenschaftlicher, politischer und gesellschaftlicher Diskurse. In diesem Kapitel wird beispielhaft aufgezeigt, wie der entwickelte Programmbegriff für diese Diskurse den Gegenstand des Programms anhand seiner Erfahrungszugänge strukturieren kann. Auf diesem Weg wird die Anwendbarkeit und der analytische Mehrwert des Begriffs für die Diskurse demonstriert.

Im Rahmen dieser Demonstration kann der Begriff gesellschaftlicher Normen weit gefasst werden, so dass er Konventionen, Umgangsformen und ungeschriebene Verhaltenserwartungen, aber auch vertragliche Übereinkünfte, Lizenzbedingungen und Gesetze mit umfasst. Gemeinsamkeit dieser Normen und der ethischen Überlegungen ist, dass die Bewertung von Handlungen darin eine zentrale Rolle spielt. Der Programmbegriff ist dafür insofern relevant, wie die Gegenstände, die mit den Handlungen in Verbindung stehen, einen Unterschied machen. Werden zum Beispiel Handlungen bewertet, die herbeigeführt haben, dass ein Webserver nicht mehr funktioniert, so sind diese Handlungen auf ein Programm bezogen. Dieser Bezug ist jedoch unterschiedlich, je nachdem, ob eine physische Manipulation vorliegt, Quellcode verändert wurde oder eine Sicherheitslücke im Webserver selbst oder einem damit verbundenen Programm in der Einbettung ausgenutzt wurde, um ihn zum Absturz zu bringen. Die Gemeinsamkeit im Programmbezug und die unterschiedlichen Gegenstandsformen, in denen das Programm darin auftritt, können auch für Diskurse um die Bewertung der Handlungen relevant sein.

In vielen Bereichen von Normen werden bereits Unterscheidungen getroffen, die große Parallelen zu den hier entwickelten Begriffsdimensionen des Programmbegriffs aufweisen. So sind rechtliche Unterscheidungen zwischen einem Eigentum an physischen Gegenständen und an Immaterialgütern längst Teil der Rechtspraxis, auch wenn gesellschaftliche Aushandlungsprozesse diesen Bereich immer wieder berühren. Der entwickelte Programmbegriff zeigt nun neben der Unterscheidbarkeit seiner Begriffsdimensionen insbesondere deren Zusammenhänge miteinander auf. Über die Zusammenhänge der Gegenstände sind auch die Normen miteinander verknüpft. Erst durch ein Zusammendenken der Begriffsdimensionen kann z.B. der Erwerb eines Computerprogramms als sinnvolle Handlung verstanden werden: Er umfasst den physischen Besitz eines Datenträgers (direkt oder über eine in der Abwicklung verfügbar gemachte Kopie, z.B. als Download), der wiederum das Programm in einer syntaktischen Form speichert (bei kommerzieller Software häufig als Maschinencode), sowie Nutzungsrechte, also unter anderem Normen bezüglich der Ausführung dieses syntaktischen Programms. Durch die unterschiedliche rechtliche Behandlung umfasst die Erwerbshandlung häufig mehr als eine Vereinbarung, der zugestimmt werden muss:

> „Wenn heute ein Kunde im Einzelhandel ein Computerprogramm erwirbt, schließen die beiden Parteien einen Kaufvertrag ab. Wenn er zuhause die Schachtel öffnet oder spätestens, wenn er beginnt, die Software zu installieren, wird er damit konfrontiert, eine Lizenz zu akzeptieren. Er muss also in einen zweiten Vertrag einwilligen, diesmal mit dem Softwareverlag, nicht für das Werkstück – das besitzt er ja bereits –, sondern für die Nutzung des darin enthaltenen Werkes. Beim unkörperlichen Vertrieb von Software über das Internet

entfällt der erste Schritt. Auf die gleiche Weise weden zunehmend auch andere digitale Werkarten lizenziert.

In diesen Lizenzen muss der Nutzer sich häufig verpflichten, zusätzliche Bedingungen zu erfüllen und auf Rechte zu verzichten – z.B. das Programm zu dekompilieren –, die ihm nach Copyright- und Urheberrecht zustehen. Lizenzen, wie die konventionelle Softwareindustrie sie verwendet, verbieten die Nutzung durch Dritte, das Kopieren, die Weiterverbreitung und die Modifikation.“[158]

Eine ganze Reihe gesellschaftlicher Normen betreffen unmittelbar syntaktische Programme, also den Quellcode oder ausführbaren Maschinencode. Die genannten Lizenzvereinbarungen, Copyright und Urheberrecht sind Beispiele für rechtlich bindende Regeln darüber, was mit diesem Gegenstand gemacht werden darf oder nicht gemacht werden darf. Programme, bei denen diese Regeln eng gesetzt sind, deren Weiterverbreitung und Modifikation rechtlich oder technisch stark eingeschränkt wird, werden als *proprietäre Software* bezeichnet. In einer sehr weiten Auslegung umfasst dies jedes Programm, das nicht entsprechend den Normen für *Freie Software* (siehe unten) lizenziert und weitergegeben wird. Die Unterscheidung zwischen diesen beiden Konzepten bezieht sich dabei ausdrücklich auf die Rechte an der Nutzung und am Quellcode, sie erfolgt unabhängig davon, ob die Software zu kommerziellen Zwecken entwickelt und vertrieben wird.

Der zentrale Unterschied zwischen proprietärer und Freier Software liegt daher auch nicht in der Verfügbarkeit oder der Weitergabe in Form von Quellcode, sondern in den Lizenzvereinbarungen, die zur Nutzung der Programme getroffen werden (auch wenn diese Lizenzvereinbarungen mithin die Verfügbarkeit und Weitergabe betreffen). Freie Software zeichnet sich dadurch aus, dass sie bestimmte Grundfreiheiten einräumt: Diese sind (erstens) die Freiheit, die Software auszuführen und zu beliebigen Zwecken anzuwenden, (zweitens) die Freiheit, den Quellcode einzusehen und zu modifizieren, (drittens) die Freiheit, die Software zu kopieren und weiterzugeben und (viertens) die Freiheit, die Software in modifizierter Form zu kopieren und weiterzugeben[159].

Ein häufig mit Freier Software verbundenes Konzept ist *Copyleft*. Copyleft beschreibt eine Eigenschaft der Lizenzen, die die Freiheit zur Weitergabe (auch in modifizierter Form) insofern einschränkt, dass sie unter einer Lizenz erfolgen muss, die die gleichen Rechte einräumt, also kompatibel sein muss. Programme, die Quellcode aus Copyleft-lizenzierter Software verwenden, müssen also selbst Copyleft-lizenziert sein. Freie Software, die nicht Copyleft-lizenziert verbreitet wird, darf dagegen auch in proprietärer Software verwendet werden. Eine schwache Form des Copyleft-Konzepts bezeichnet eine Lizenzierung von Programmbibliotheken derart, dass die reine Anwendung der Bibliothek auch durch nicht Copyleft-lizenzierte Software erfolgen darf, die Verbreitung der Bibliothek selbst (auch modifizierter Versionen) jedoch wiederum unter Copyleft-Lizenzen stattfinden muss. Software unter *permissiven* Lizenzen, die eine Verwendung auch in proprietärer Software gestatten, kann auch Freie Software sein; Copyleft stellt eine zusätzliche Eigenschaft dar.

Einer der Hauptakteure im Umfeld Freier Software ist die Free Software Foundation (FSF). Diese Stiftung fördert Freie Software insbesondere durch Entwicklung von Lizenzvereinbarungen, die die genannten Grundfreiheiten einräumen, und die rechtliche und

158 Grassmuck, Volker: Freie Software. Zwischen Privat- und Gemeineigentum. Bonn. Bundeszentrale für politische Bildung, 2002. S. 122f.

159 Zur Definition Freier Software nach der Free Software Foundation siehe: https://www.gnu.org/philosophy/free-sw.html (abgerufen am 18.09.2018).

strukturelle Unterstützung von Softwareentwicklung in diesem Bereich. Die von der FSF unterstützten Lizenzen umfassen dabei für Software die weit verbreitete GPL (GNU General Public License, eine Lizenz mit starkem Copyleft) und LGPL (GNU Lesser General Public License, eine Lizenz mit schwachem Copyleft). Im Selbstverständnis der FSF ist Freie Software ein ethisches und politisches Unterfangen und notwendig, um grundlegende Freiheiten zu schützen. Proprietäre Software wird als grundsätzlich unethisch abgelehnt:

> „To use free software is to make a political and ethical choice asserting the right to learn, and share what we learn with others. Free software has become the foundation of a learning society where we share our knowledge in a way that others can build upon and enjoy.
>
> Currently, many people use proprietary software that denies users these freedoms and benefits. If we make a copy and give it to a friend, if we try to figure out how the program works, if we put a copy on more than one of our own computers in our own home, we could be caught and fined or put in jail. That's what's in the fine print of the license agreement you accept when using proprietary software.
>
> The corporations behind proprietary software will often spy on your activities and restrict you from sharing with others. And because our computers control much of our personal information and daily activities, proprietary software represents an unacceptable danger to a free society."[160]

Die Argumente für Freie Software bestehen in diesem Auszug des Selbstverständnisses der FSF aus zwei wesentlichen Teilen. Der erste Teil betont die gesellschaftlichen Vorteile durch das Teilen von Wissen. Dieses Argument zielt darauf ab, dass die freie Verfügbarkeit von Wissen positive gesellschaftliche Entwicklungen zur Folge habe und dass der kooperative Prozess, in dem Freie Software zustande kommt, selbst als wesentliche Freiheit (zur Kooperation) geschützt und unterstützt werden solle. Der zweite Teil drückt die Ablehnung proprietärer Software aus, da durch die Verwendung dieser die Kontrolle über den eigenen Computer an die Anbieter der Software abgegeben werde – und durch die fehlende Möglichkeit, den Quellcode kollektiv oder individuell zu überprüfen und zu ändern, sich diese Kontrolle, die sich auf alle gespeicherten Daten und jede denkbare computerunterstützte Aktivität ausweiten könne, letztendlich in Überwachung und Unterdrückung niederschlagen könne[161]. In diesem Zusammenhang wird proprietäre Software als Gefahr für eine freie Gesellschaft verstanden.

Die Programme werden dabei in unterschiedlichen Perspektiven aufgefasst: Der erste Teil zielt auf syntaktische und semantische Programme ab. Das Wissen, das geteilt werden soll, ist im Quellcode der Programme festgehalten. Die Kooperation zielt insbesondere auf die Zusammenarbeit beim Programmieren ab, und auch die Möglichkeit zur Weitergabe von Programmen unter den Lizenzen Freier Software bezieht sich auf eine Weitergabe syntaktischer Programme. Der zweite Teil dagegen bezieht ausgeführte Programme mit ein. Die Möglichkeit der illegitimen Kontrolle von Anwender_innen und deren Computer durch die Anbieter von proprietärer Software bezieht sich gerade auf die Einbettung, in der Programme angewendet werden. Das Verhältnis des laufenden Programms zu dem System, in das es eingebettet ist, ist hier dadurch gekennzeichnet, dass eine Abgabe der Kontrolle über das

160 https://www.fsf.org/about/what-is-free-software (abgerufen am 18.09.2018).

161 Zum Selbstverständnis des im Wesentlichen von der FSF unterstützten GNU-Projekts siehe die Seiten unter https://www.gnu.org/philosophy/philosophy.html (abgerufen am 18.09.2018).

Programm eine Abgabe der Kontrolle über das gesamte System zur Folge hat. In dieser Hinsicht sieht die FSF nicht nur die unmittelbar mit proprietärer Software verknüpften Daten durch diese in Gefahr, sondern den gesamten Computer inklusive der gespeicherten Daten und dessen jeweilige Einbindung in andere Handlungszusammenhänge. Hierbei fällt die enge Verknüpfung vom eingebetteten Programm mit dem Quellcode auf – Vertrauen in ein Programm zur Laufzeit setzt in der Perspektive der FSF zwingend ein Vertrauen in den Quellcode voraus, das nur durch eine direkte Zugangs- und Kontrollmöglichkeit erzeugt werden kann.

Ein anderes normatives Konzept, unter dem die Einräumung wesentlicher auf den Quellcode bezogener Freiheiten verstanden wird, ist Open Source Software. Hierbei ist wichtig, dass der Begriff *Open Source* zunächst als Rebranding konzipiert wurde, um wirtschaftsfeindliche Assoziationen und Stereotype, die mit dem Begriff Freier Software zusammenhingen, zu überwinden. Ziel dabei war es, innerhalb kurzer Zeit durch eine Marketingkampagne für quelloffene Software diese als taugliches gewinnorientiertes Geschäftskonzept in der Softwareindustrie, insbesondere im Management und bei Investoren, zu etablieren[162]. Dadurch ist nicht nur die Software, die als Open Source bezeichnet wird, weitgehend deckungsgleich mit freier Software – auch die Anforderungen an diese, die durch die Open Source Initiative (OSI) als zentralem Akteur in der *Open Source Definition*, einem Standard für Lizenzen, festgelegt werden, entsprechen den Anforderungen an Freie Software, die in den vier genannten Freiheiten ausgedrückt ist. Dadurch sind fast alle häufig verwendeten Softwarelizenzen, die unter die Open Source Definition fallen, gleichzeitig von der FSF als Lizenzen für Freie Software aufgelistet[163]. Die Extension von Freier Software und Open Source Software ist praktisch gleich – lediglich vereinzelt werden Lizenzen von der OSI als Open Source aufgeführt, aber von der FSF nicht als Freie Software anerkannt[164].

Allerdings betont die OSI andere Aspekte und andere Begründungen für die Unterstützung quelloffener Software als die FSF, und dieser Unterschied in der Auffassung von Software liegt unter anderem in der Geschichte des Open Source-Begriffs begründet, in der die Entwicklung quelloffener Software zu kommerziellen Zwecken als rationale Strategie auch für große Unternehmen dargestellt werden sollte. Dadurch werden im Selbstverständnis der OSI und ihren Aktivitäten insbesondere die Vorteile quelloffener Software für die Softwareentwicklung selbst hervorgehoben:

> „We are also actively involved in Open Source community-building, education, and public advocacy to promote awareness and the importance of non-proprietary software. OSI Board members frequently travel the world to attend Open Source conferences and events, meet with open source developers and users, and to discuss with executives from the public and private sectors about how Open Source technologies, licenses, and models of development can provide economic and strategic advantages.[...]
>
> Open source enables a development method for software that harnesses the power of distributed peer review and transparency of process. The promise of open source is higher quality, better reliability, greater flexibility, lower cost, and an end

162 Die Geschichte dieses Begriffs, die eng mit der Firma Netscape Communications und ihrer Konkurrenz zu Microsoft im Browserbereich verwoben ist, reflektiert Raymond als Beteiligter in Raymond, Eric S.: The Cathedral and the Bazaar. Musings on Linux and Open Source by an Accidental Revolutionary. O'Reilly, 2001. S. 175ff.

163 Vgl. https://www.gnu.org/licenses/license-list.html#SoftwareLicenses (abgerufen am 18.09.2018). und https://opensource.org/licenses/category (abgerufen am 18.09.2018).

164 Z. B. die *Sybase Open Watcom Public License version 1.0* Siehe https://www.gnu.org/philosophy/open-source-misses-the-point.en.html (abgerufen am 18.09.2018).

to predatory vendor lock-in.“[165]

Die Argumentationsstruktur der OSI ist konsequenzialistisch: Quelloffene Software ist in dieser Perspektive wünschenswert, weil der offene Prozess effizienter sei und Software von höherer Qualität, Flexibilität und Zuverlässigkeit hervorbringe. Lediglich der am Ende genannte Lock-in-Effekt, also das Ausnutzen von Pfadabhängigkeit und hoher Kosten für ein Austauschen der verwendeten Software, ist mit dem Argument der Kontrolle der FSF verknüpft. Allerdings ist auch dieses Argument bei der OSI durch die Lock-in-Terminologie ein wirtschaftliches, während die FSF die Freiheit der Anwender_innen in den Vordergrund stellt. Auch das Lock-in-Argument zielt darauf ab, die Entwicklung quelloffener Software als wirtschaftlich vernünftige Entscheidung zu legitimieren. So wird insgesamt die Konklusion unterstützt, dass quelloffene Entwicklungsmodelle den geschlossenen Modellen proprietärer Software in vielen Hinsichten überlegen sind. Diese Überlegenheit wird dagegen in der Perspektive der FSF nicht als zentraler Punkt gesehen. So schreibt Richard M. Stallman (Gründer der FSF und des GNU-Projekts zur Entwicklung eines freien Betriebssystems, zentraler Akteur in der FSF):

> „The idea that we want software to be powerful and reliable comes from the supposition that the software is designed to serve its users. If it is powerful and reliable, that means it serves them better.
>
> But software can be said to serve its users only if it respects their freedom. What if the software is designed to put chains on its users? Then powerfulness means the chains are more constricting, and reliability that they are harder to remove. Malicious features, such as spying on the users, restricting the users, back doors, and imposed upgrades are common in proprietary software, and some open source supporters want to implement them in open source programs. […]
>
> This software might be open source and use the open source development model, but it won't be free software since it won't respect the freedom of the users that actually run it. If the open source development model succeeds in making this software more powerful and reliable for restricting you, that will make it even worse.“[166]

Diese unterschiedlichen Perspektiven auf quelloffene Software durch die FSF und die OSI kommen mit unterschiedlichen Begründungen jeweils zu der Bewertung, dass quelloffene Software besser sei als proprietäre Software. Diese unterschiedlichen Begründungen können unter Verwendung des entwickelten Programmbegriffs dahingehend untersucht werden, ob der Zugang zu Software bzw. zu Programmen sich dabei unterschiedlich darstellt. Eine solche Analyse eröffnet neue Perspektiven auf die normativen Diskurse, die bezüglich Programmen geführt werden.

Zunächst ist hervorzuheben, dass beide Perspektiven die Zugänglichkeit zum Quellcode auch physisch fassen. Sowohl die Open Source Definition als auch die Definition Freier Software über die vier Grundfreiheiten fordern Möglichkeiten, die Software in physischer Form zu erhalten. Dies kann über den Versand eines Datenträgers stattfinden (die FSF wurde 1985 gegründet, Internetzugänge für Privatnutzer waren nicht weit verbreitet), oder durch die Möglichkeit zum Download der Quellcodedateien. Weiter fordern beide

165 https://opensource.org/about (abgerufen am 18.09.2018).

166 https://www.gnu.org/philosophy/open-source-misses-the-point.en.html (abgerufen am 18.09.2018).

Definitionen die Verfügbarkeit in menschenlesbarer Form – das Programm soll in der syntaktischen Form weitergegeben werden, in der es auch entwickelt wurde. Die Weitergabe ausschließlich in übersetzter Form, wie sie für proprietäre Software üblich ist, wird ausgeschlossen. Dies zeigt, dass die maschinelle Semantik für die Abgrenzung von Freier Software und Open Source Software zu proprietärer Software gerade nicht entscheidend ist – sondern der Zugang zur Semantik einschließlich der natürlichsprachlichen Semantik über das syntaktische Programm.

Ein wesentlicher Unterschied liegt in der Form, in der das Verhältnis von Quellcode zum ausgeführten Programm gefasst wird. Bei Open Source liegt die Betonung auf der Qualität – in den Argumenten der OSI wird behauptet, dass quelloffene Software in der Ausführung ihre Funktion besser erfüllen könne als proprietäre Software. Bei Freier Software dagegen steht das Verhältnis von Kontrolle und Freiheit im Vordergrund: Nur durch die durch Quelloffenheit gegebene Kontrolle über das Programm durch die Anwender_innen könne die Kontrolle über das ausführende System sichergestellt werden, und nur durch die Freiheit zur Modifikation könne wiederum die Freiheit der Anwender_innen geschützt werden, nicht umgekehrt durch die Software kontrolliert zu werden. Durch diesen Bezug auf Freiheit kommt den Anwender_innen in der Perspektive der FSF eine deutlich aktivere Rolle zu: Ihrer Möglichkeit, den Quellcode selbst zu inspizieren und zu modifizieren und dadurch die Kontrolle über die Technik zu behalten, steht in der Perspektive der OSI lediglich der passive Vorteil gegenüber, bessere und effizienter entwickelte Programme zu erhalten.

Der Unterschied der Perspektiven im Zugang zu Programmen wird insbesondere im Argument der FSF deutlich, Freie Software biete gesellschaftliche Vorteile durch die kooperative Produktion und das Teilen von Wissen. In diesem Argument spielt das eingebettete Programm keine Rolle, stattdessen wird das Teilen eines syntaktischen Zugangs zu einem semantischen Programm deswegen positiv bewertet, weil dieses Teilen eine Weitergabe von Wissen darstellt: Andere können von einem Programm lernen, Problemlösungen übernehmen und darauf aufbauend selbst neue Programme, neues Wissen produzieren. Die Veröffentlichung des Quellcodes, also einer Programmsemantik in menschenlesbarer syntaktischer Form, stellt in dieser Argumentation einen maßgeblichen Schritt dar, der technisches Wissen allen zugänglich machen soll – von diesem Wissen soll in der Perspektive der FSF niemand durch technische oder rechtliche Hürden ausgeschlossen werden können. Eine praktische Konsequenz aus dieser Verknüpfung quelloffener Software mit kooperativer Wissensproduktion kann in der starken Unterstützung der FSF von Copyleft-Lizenzen gesehen werden: Die gesellschaftlichen Vorteile des öffentlichen Zugangs zu Wissen summieren sich, wenn dieses Wissen wiederum zur Produktion weiteren öffentlich zugänglichen Wissens verwendet wird[167].

Dieser freie Zugang zu Wissen, das im Quellcode von Programmen ausgedrückt ist, sowie die Freiheit zur Modifikation und zur Veröffentlichung modifizierter Programme hat einen eigenen Umgang mit Wissen und mit der Strukturierung dieses Wissens (z.B. in verschiedenen Versionen und Varianten von Programmen) hervorgebracht. Eric S. Raymond fasst diesen durch die Lizenzen festgeschriebenen Zugang mit *anyone can hack anything* zusammen. Raymond untersucht in seinem Aufsatz *Homesteading the Noosphere*, inwiefern es in den kooperativen Umgangsformen einer *Hackerkultur* trotz dieser lizenzrechtlichen Rahmenbedingungen, in der jede Modifikation erlaubt ist, Zuschreibungen von Besitz an Quellcode gibt (OSD steht hier für *Open Source Definition*):

167 Dies ist ein positives Argument für Copyleft-Lizenzen. Eine Variante hiervon ist, dass Copyleft-Lizenzen weiterer Freier Software wettbewerbliche Vorteile bieten, da diese auf Copyleft-lizenzierte Bibliotheken zurückgreifen können, und so ein Anreiz zur Entwicklung weiterer Freier Software entsteht. Ein negatives Argument gegen Lizenzen ohne Copyleft aus dem Umfeld Freier Software ist, dass dadurch die als unethisch empfundene unfreie Software unterstützt werden könne.

„Under the guidelines defined by the OSD, an open-source license must protect an unconditional right of any party to modify (and redistribute modified versions of) open-source software.

Thus, the implicit theory of the OSD (and OSD-conformant licenses such as the GPL, the BSD license, and Perl's Artistic License) is that anyone can hack anything. Nothing prevents half a dozen different people from taking any given open-source product (such as, say the Free Software Foundations's gcc C compiler), duplicating the sources, running off with them in different evolutionary directions, but all claiming to be the product. […]

The open-source licenses do nothing to restrain forking, let alone pseudo-forking; in fact, one could argue that they implicitly encourage both. In practice, however, pseudo-forking is common but forking almost never happens. Splits in major projects have been rare, and are always accompanied by re-labeling and a large volume of public self-justification. […]

In fact (and in contradiction to the anyone-can-hack-anything consensus theory) the open-source culture has an elaborate but largely unadmitted set of ownership customs."[168]

Open Source und Freie Software prägen so eine Hackerkultur, die eigene gesellschaftliche Normen bezüglich Besitz und Eigentum hervorbringen. Die Verknüpfung der Besitzzuschreibung mit dem Programm ist hier weitestgehend auf die strukturelle Einbettung desselben bezogen: Die Besitzzuschreibung findet auf langfristige Entwicklungsprojekte bezogen statt. Hier ist jeweils ein Mensch oder eine Gruppe von Menschen maßgeblich dafür verantwortlich, richtungsweisende Entscheidungen für die Entwicklung zu treffen. Diese Entscheidungen beziehen sich nicht auf eine bestimmte Version des syntaktischen oder semantischen Programms, sondern auf die Funktionalität, die es in Verbindung mit seiner Einbettung erfüllen soll. Entwickelt zum Beispiel ein solches Projekt einen quelloffenen Compiler (weiter), so ist der Gegenstand des Projekts durch die Bedeutung des Compilers in seinen Anwendungszusammenhängen gegeben (als Übersetzer von Sprache X in Sprache Y). Die Entwicklungsarbeit und damit verbundene Prozesse wie z.B. Bugtracking werden maßgeblich durch ein zentrales Team von Entwickler_innen koordiniert. Abweichende Versionen durch andere, z.B. Forks, sind zwar legal, spielen aber in der Praxis keine große Rolle. Eine solche Fassung von Besitz ist eng mit Verantwortung verknüpft: Sollte die Funktionalität nicht mehr den Ansprüchen der Anwender_innen genügen, kann durch die Quelloffenheit schnell ein Parallelprojekt entstehen, das ausgehend vom zugänglichen Quellcode die Anforderungen zu erfüllen versucht. Nur durch die verantwortliche Koordination wird der „Besitz" akzeptiert. Die Besitzzuschreibung könnte also auch folgendermaßen ausgedrückt werden: Diese Gruppe ist für die Entwicklung dieses Compilers *verantwortlich*. Ein Beispiel, in dem aufgrund dieser Verantwortungszuschreibung der „Besitz" eines Projekts gewechselt wurde, ist die Programmbibliothek uClibc: Der Fork uClibc-ng wurde als relevant akzeptiert, nachdem das ursprüngliche Projekt „verwaist" war; es erfolgten über mehrere Jahre keine Aktualisierungen und der Verantwortliche war nicht erreichbar[169].

168 Raymond, Eric S.: The Cathedral and the Bazaar. Musings on Linux and Open Source by an Accidental Revolutionary. O'Reilly, 2001. S. 71f.

169 Siehe http://lists.uclibc.org/pipermail/uclibc/2014-July/048400.html (abgerufen am 20.10.2018).

4.4 Exkurs: Programme und Zeitlichkeit

Mit Computerprogrammen, ihrer Entwicklung und ihrer Anwendung sind viele Vorstellungen von Zeit und, eng mit der Zeit verbunden, von Geschwindigkeit verknüpft. Zeitliche Eigenschaften der Gegenstände sowie ihre Veränderung über die Zeit werden im hier gewählten phänomenologischen Zugang den Gegenständen im Akt der Wahrnehmung, in ihrer Konstitution zugeschrieben: Wird ein Programm als „neu“ oder „schnell“ konstituiert, so kann sich ein syntaktisch gleiches Programm in einer anderen Wahrnehmung (z.B. nachdem eine neue Version veröffentlicht wurde) als „alt“ oder „langsam“ darstellen. Der in dieser Arbeit entwickelte Programmbegriff eignet sich dazu, diese Zeitvorstellungen nach der Form der Gegenstände, auf die sie bezogen sind, einzuordnen.

Mit Zeit und Geschwindigkeit können dabei zunächst physische Eigenschaften beschrieben werden, als physisch konstituierte Programme (z.B. auf einem Datenträger) werden zu einer bestimmten Zeit an einem bestimmten Ort wahrgenommen, sie haben eine physisch gedachte Geschwindigkeit, die sich in zurückgelegtem Weg über eine bestimmte Zeit ausdrücken lässt. Diese physischen Eigenschaften sind im Bezug zu Programmen jedoch häufig gerade nicht gemeint: Auch eine „langsame“ Festplatte beschleunigt unter Bedingungen oberflächennaher Erdanziehung mit ca. 9,81 m/s^2, sie wird dadurch allerdings nicht „schneller“. Und wenn von einer Verbindung über WLAN auf das „schnellere“ Glasfaserkabel umgestiegen wird ist mit „schnell“ nicht das Signal gemeint – tatsächlich breiten sich die Lichtwellen im Glasfaserkabel schon aufgrund der unterschiedlichen Brechungsindizes von Luft und Glas langsamer aus als die Funkwellen des WLAN. Die „alte“ Version eines Programms bezieht sich nicht auf die physische Zeit, zu der sie kopiert wurde oder zu der der Datenträger hergestellt wurde, „veraltet“ ist sie binnen weniger Minuten oder vieler Jahre – das Alter kann hier offensichtlich nicht ausschließlich physisch verstanden werden.

Zeit wird in der menschlichen Erfahrung in vielen verschiedenen Formen konstituiert. Sie wird als subjektives Erleben der Erinnerung und der Erwartung erfahren, und als Erleben der irreversiblen Veränderung der als unabhängig vom Selbst wahrgenommenen Gegenstände, als Weltzeit konstituiert. Insbesondere aber wird Zeit auch gesellschaftlich geprägt, ihre Messbarkeit und Bedeutung in Verbindung zu anderen sozialen Institutionen und Normen gestaltet. Die Möglichkeit, Zeit in festen Größen von Sekunden, Tagen oder Jahren anzugeben, unterscheidet sich sowohl vom subjektiven Empfinden von Zeit als auch von der Erfahrung, dass die wahrnehmbare Welt Veränderungen unterworfen ist. Diese unterschiedlichen Bedeutungen von Zeit sind eng miteinander verknüpft, sie prägen die Konstitution von Gegenständen in der Zeit zusammen:

> „Das Wissen um die Endlichkeit hebt sich von der Erfahrung der Fortdauer der Welt ab und ist das von der lebensweltlichen Zeit bestimmte Grundmoment aller Entwürfe im Rahmen des Lebensplans. In die Verwirklichung konkreter Entwürfe, in die alltägliche Lebensführung treten aber als bestimmende Faktoren noch weitere, mit der Struktur der lebensweltlichen Zeit verknüpfte Momente. Die Struktur der lebensweltlichen Zeit baut sich auf in Überschneidungen der subjektiven Zeit des Bewußtseinsstroms, der inneren Dauer, mit der Rhytmik des Körpers wie der »biologischen Zeit« überhaupt, mit den Jahreszeiten wie der Welt-Zeit überhaupt und dem Kalender, der »sozialen Zeit«. Wir leben in all diesen Dimensionen zugleich.“[170]

170 Schütz, Alfred und Luckmann, Thomas: Strukturen der Lebenswelt. Band 1. Suhrkamp, 1979. S. 75.

Im Bezug auf Computerprogramme ist insbesondere die von Schütz und Luckmann genannte soziale Zeit von Bedeutung: Der zeitliche Abstand zwischen zwei Taktsignalen eines Prozessors, die geschätzte Dauer einer komplexen Berechnung, die Dauer eines Entwicklungszyklus (*Sprint*) im Scrum-Prozess oder die Dauer der Langzeitunterstützung einer Linux-Distribution werden in gesellschaftlich stabilisierten Einheiten (wie Sekunden oder Jahren) gemessen und in Aushandlungsprozessen mit Bedeutung gefüllt. Durch die Festlegung von Messinstrumenten und Einheiten kann einem Prozessor die Taktzykluslänge von 0,03 Mikrosekunden zugeschrieben werden, und in einem weiteren Schritt kann diese Größe unter soziotechnischen Rahmenbedingungen als extrem lange Zeit für einen gegenwärtigen Prozessor aufgefasst werden (Die Zykluslänge für aktuelle Prozessoren in PCs und Laptops liegt üblicherweise unter einer Nanosekunde).

Das Beispiel zeigt, dass Zeit (und, in Verbindung mit ihr, Geschwindigkeit) häufig in Relationen, z.B. in Vergleichen, wahrgenommen und mit Bedeutung gefüllt wird. 30 Nanosekunden liegen weit unterhalb einer für Menschen wahrnehmbaren Zeitspanne, als lange Zeit können sie sich nur im Vergleich z.B. zu anderen Zykluslängen darstellen. Eine Scrum-Sprintdauer von 14 Tagen kann als kurz bezeichnet werden, während eine Programmlaufzeit von 14 Tagen für eine Berechnung in vielen Zusammenhängen als sehr lange gelten kann. Die Relationen, in denen zeitliche Eigenschaften wahrgenommen und kommuniziert werden, sind dabei sowohl von der genauen Charakterisierung des Gegenstands als auch von den Rahmenbedingungen abhängig. Das zeitliche Verhalten eines Mikrocontrollers wird in anderen Relationen gesehen und bewertet als das eines Smartphone-Prozessors, dieses wiederum anders als das eines Prozessors für Desktop-PCs. In allen drei Fällen ist die Bewertung als „langsam“ oder „schnell“ vom aktuellen Stand der Prozessortechnologie abhängig.

Viele Konzepte im Umfeld von Programmen sind auf die Zukunft bezogen. So sind die Gestaltung von Produktlebenszyklen und das Releasemanagement, die Automatisierung von Tests und allgemein Sofwareentwicklungsprozesse, die Betrachtung der Wartbarkeit von Software und das Innovationsmanagement auf Erwartbarkeit und Planbarkeit von zukünftigen Entwicklungen bezogen. Helga Nowotny analysiert in *Eigenzeit*[171] diese Ausweitung des Planungshorizonts, in der die Zukunft weniger als Ungewissheit und dafür stärker als Folge der Gegenwart betrachtet wird. Die Zukunft als Projektionsfläche für gegenwärtig nicht Gegebenes, sei es in der Furcht oder in der Hoffnung, wird durch die Auffassung als Folge planbarer Handlungen abgelöst durch eine *erstreckte Gegenwart*. Diese Form von Gegenwart macht die Zukunft verfügbar, indem sie mit Planungen und Optionen umgeht, deren Folgen jeweils antizipiert werden. Diese Ausweitung der Gegenwart hat unter anderem Folgen für das Verhältnis zwischen linearen Zeitvorstellungen (in denen jedes Geschehen einzigartig ist) und zyklischen Zeitvorstellungen (in denen Wiederholungen zentral sind, z.B. im Tages- oder Jahresrhythmus, aber auch auf dem Ziffernblatt einer Uhr):

> „Doch die Abschaffung der Kategorie der Zukunft und ihre Ersetzung durch die der erstreckten Gegenwart reicht nicht aus, um mit allen aufgetretenen Widersprüchen und dem Problemdruck fertig zu werden. In Bewegung geraten ist vielmehr auch das Verhältnis zwischen der linearen und der zyklischen Zeitauffassung. Dieses wird im allgemeinen als grundlegend für die »zeitliche Architektur einer Zivilisation« (Pomian) angesehen. Sie kann in ihren Bestandteilen auseinandergenommen und wieder zusammengesetzt werden, doch selbst durch die Pluralität der Zeiten – der psychologischen Zeit, der Sonnen-, liturgischen und politischen Zeit, der Zeit der Uhren und der ultralangen oder

171 Nowotny, Helga: Eigenzeit. Entstehung und Strukturierung eines Zeitgefühls. Suhrkamp. 1995 (Erstauflage 1993).

> extrem kurzen Zeit der Wissenschaft [...] – zieht sich der tief verwurzelte Dualismus des Denkens und der Zeitwahrnehmung in geraden Linien oder in wiederkehrenden Bewegungen. Die eigene Lebenserfahrung bietet beiden Anschauungsweisen genügend Nahrung. Die Bewegung von der Geburt bis zum Tod kann als Teil eines Kreislaufs, ja als Wiederkehr, gedacht werden oder als eine mehr oder weniger gerade Linie, die vom Ausgangspunkt zum Endpunkt führt. Das Lineare, meint M. Young, betont das Neue und läßt Bekanntes, bereits Erlebtes hinter sich. So gesehen wiederholt es sich nicht und nichts wird in ihm wiederholt. Doch ebensowenig kann das Zyklische, das Bekannte, schon einmal Gewesene, im menschlichen Leben Identisches wiederbringen, obwohl es Neues produziert. [...] Wie jeder Dualismus, findet sich reichliches Anschauungsmaterial für beide Sehweisen, doch sie bleiben in der Bewertung, die ihnen aus sozialer Warte zukommt, in eigentümlich ambivalenter, instabiler, weil leicht »umwertbarer« Position. Die erstreckte Gegenwart, die nicht in einer linear offenen, fortgesetzten Zukunft mehr gedacht werden kann, muß daher zyklisches Zeitgut notwendigerweise neu evaluieren und in sich aufnehmen.“[172]

Auf Computerprogramme bezogen lässt sich diese Relevanz zyklischer Zeitvorstellungen für die Gegenwart, die bereits in der Planung Vorstellungen von Zukunft in sich aufnimmt, an Beispielen leicht aufzeigen. Die zyklischen Vorstellungen beziehen dabei das Neue auf das Bekannte, das schon Dagewesene, zurück. So stellt sich in der Planung von Entwicklungsprozessen der nächste Scrum-Sprint als gleichförmig zum letzten dar. Es werden zwar andere Funktionen implementiert, andere Fehler behoben und anderer Quellcode geschrieben, allerdings bleibt die Form des Entwicklungszyklus gleich. Aufwandsbewertungen, z.B. in Form von *Story Points* (Eine Bewertung des Programmieraufwands zur Implementierung der Funktionalität einzelner User Stories), sollen die Zyklen gleichförmig und planbar in der Menge implementierter Funktionalität machen – die Quantifizierung von Aufwand ist auf die Zukunft bezogen, Geschwindigkeit wird zyklisch in Punkten pro Sprint angegeben. Der Releasezyklus einer Linux-Distribution soll die Softwareanwendung in der Zukunft planbar machen, wenn Langzeitunterstützung (z.B. Sicherheitsupdates) für mindestens 5 Jahre ab Release zugesichert wird.

Das Verhältnis von linearen zu zyklischen Zeitvorstellungen ist dadurch gekennzeichnet, dass in vielen Bereichen beide Vorstellungsformen zusammen auftreten und miteinander verknüpft sind. Der Satz „Der Zug fährt morgen später.“ wäre ohne die Verknüpfung beider Vorstellungen tautologisch (in einer rein linearen Vorstellung) oder unverständlich (wenn lediglich die zyklische Gleichförmigkeit der Tage betrachtet wird). Durch die Verknüpfung beider Vorstellungen transportiert er jedoch eine verständliche und möglicherweise hilfreiche Information. Auch in Programmen sind die Vorstellungen häufig verknüpft: Eine Schleife wiederholt zwar in ihrer Ausführung zyklisch Instruktionen, ist jedoch zeitliche Linearität in keiner Weise damit verknüpft (z.B. in den verarbeiteten Daten, in Variablen oder auch über ein Eingabegerät), dann stellt sich die Schleife als Endlosschleife dar, meist ungewollt und damit als Programmfehler. Der Dualismus von Linearität und Zyklizität der Zeit kann also nicht als ausschließender Gegensatz erfasst werden. Barbara Adam fasst diesen Gedanken in einer Kritik der Charakterisierung von Gesellschaften anhand zyklischer und linearer Zeitvorstellungen zusammen. Die Zuschreibungen, bestimmte Gesellschaften würden sich ausschließlich in einer zyklischen bzw. linearen Zeit begreifen, sei unhaltbar:

172 Nowotny, Helga: Eigenzeit. Entstehung und Strukturierung eines Zeitgefühls. Suhrkamp. 1995 (Erstauflage 1993). S. 55f. Nowotny bezieht sich hier auf Michael Youngs *The Metronomic Society* und Krzysztof Pomians *L'ordre du temps*.

> „Statt dessen gilt es, anzuerkennen, daß alle sozialen Prozesse Aspekte von Linearität und Zyklizität aufweisen, daß wir von einer Kreisstruktur sprechen müssen, wenn wir es mit sich wiederholenden Ereignissen zu tun haben, und von Linearität, wenn wir den Prozeß der Wiederholung betrachten. Ob wir Linearität oder Zyklizität »erkennen«, ist grundsätzlich eine Frage der Perspektive und Interpretation. […] Insofern zyklische Prozesse per definitionem stets variable Wiederholungen, Linearität und Fortgang umfassen, steht hier nicht das Konzept der Zyklizität selbst zur Debatte. Das Problem liegt vielmehr in der Rolle, die der zyklischen Zeit bei der Untersuchung »anderer« Zeiten als fiktiver Gegensatz zur unreflektierten Vorstellung einer »westlichen linearen Zeit« zugewiesen wird. Wir können die unbefriedigende Dichotomie von zyklischen und linearen Aspekten der Zeit mithin auflösen, indem wir ihre fundamentale wechselseitige Implikation und ihre Relativität zur Perspektive des Betrachters betonen.“[173]

Entsprechend müssen zyklische Zeitvorstellungen auch auf Programme bezogen stets in ihrem Verhältnis zu linear gedachter Zeit betrachtet werden. Eine Schleife kann Invarianten haben, die Instruktionen mögen bei jedem Durchlauf gleich bleiben, trotzdem ist ihre Ausführung jederzeit Teil eines linearen Prozesses, beispielsweise einer Programmausführung deren Anfang und Ende benannt werden können. Scrum-Sprints sind zwar zyklisch gestaltet, das bearbeitete Programm soll jedoch zumindest syntaktisch nach jedem Sprint in einem anderen Zustand vorliegen als zuvor. In jedem Taktzyklus eines Prozessors sind dasselbe Steuerwerk und derselbe Datenpfad involviert (diese sind durch den Chipentwurf festgelegt), die ausgeführten Instruktionen, involvierten Daten und der Zustand des gesamten Systems, innerhalb dessen dem Prozessor eine Funktion zukommt, ändern sich jedoch. Die sich wiederholenden Zyklen sind immer in lineare Prozesse eingebettet. Illustrativ für dieses Verhältnis ist der Programmzähler im Prozessor: In jedem Taktzyklus (der keinen Sprung ausführt) wird er erhöht, er stellt eine der technischen Verknüpfungen linearer und zyklischer Zeitvorstellungen dar.

Mithilfe des entwickelten Programmbegriffs lassen sich die mit Programmen verknüpften Zeitvorstellungen anhand ihres Gegenstandsbezugs analysieren. So können mit den physischen Gegenständen physische Zeitvorstellungen verknüpft werden: Wird die Taktrate eines Prozessors in Takten pro Sekunde angegeben, so ist mit dieser Sekunde die gleiche Zeiteinheit gemeint, in der auch Naturphänomene oder die Ladezeit einer Batterie gemessen werden können. Die Zeit, die ein Datenpaket benötigt, um zu einem bestimmten Ziel und zurück gesendet zu werden (die *Round Trip Time*, für das Internet z.B. durch ein *Ping*-Programm gemessen), wird in physisch gemeinten Millisekunden angegeben. Über die Netzwerkadressierung wird im Internet der Ort virtualisiert, die Zeitvorstellung bleibt jedoch physisch.

Anders verhält es sich bei der Versionierung (von Dateien, insbesondere aber von Programmen): Die Versionsnummer stellt eine Virtualisierung von Zeit dar. Eine Version mit niedrigerer Versionsnummer ist *älter*, allerdings sagt dies nichts über die physisch vergangene Zeit zwischen den Veröffentlichungen. So könnte in einer Entwicklung von Version 1.0 zur Version 1.1 zur Version 1.2 der erste Schritt wenige Minuten und der zweite Jahre dauern. Versionen konstituieren lediglich ein *Vorher* und ein *Nachher*, sie sind ordinal skaliert. Physische Zeitangaben in Minuten, Jahren oder Nanosekunden sind dagegen proportional skaliert: Die Abstände zwischen 2 und 3 Sekunden sowie zwischen 7 und 8 Sekunden sind gleich, darüber hinaus existiert ein Nullpunkt. Vorher und Nachher von Versionen müssen dabei keinen einzelnen Pfad in der Zeit darstellen: So kann in einem

173 Adam, Barbara: Das Diktat der Uhr. Suhrkamp, 2005. S. 61.

Versionsverwaltungssystem mit Vorher die Menge aller Versionen gemeint sein, von der eine Version abhängig ist, mit Nachher diejenigen, die von ihr abhängig sind. Ein gerichteter azyklischer Versionsgraph ersetzt hier eine lediglich lineare Vorstellung von Zeit. Allerdings wird für die Übersichtlichkeit bei Veröffentlichungen häufig lediglich eine kleine Teilmenge der entwickelten Versionen abgebildet – diese müssen jedoch gar nicht aufeinander aufbauen, im Versionsgraph gar nicht verbunden sein. Mit der physisch gedachten Zeit ist diese Virtualisierung lediglich lose verknüpft: Jedes Vorher muss auch physisch vorher als Version im Verwaltungssystem eingecheckt worden sein. Die Versionsnummern (häufig werden Nummern ausschließlich für veröffentlichte Versionen verwendet) können Zeit gleichzeitig linear und zyklisch virtualisieren: Die aufsteigende Ordnung gibt eine lineare Reihenfolge an, während die Trennung in mehrere Zahlen Zyklen zulässt. So ist z.B. die Versionsfolge 1.0, 1.1, 1.2, 2.0 aufsteigend sortiert, die Änderung der ersten Zahl in Version 2.0 markiert als *Major Release* den Beginn eines neuen Zyklus.

Direkte Bezüge zur physischen Zeit spielen in der maschinellen Semantik von Programmen nur eine untergeordnete Rolle. So können in spezifischen Kontexten, z.B. bei der Gestaltung einer Benutzeroberfläche, physische Zeiten angegeben werden. Meist wird physische Zeit jedoch nicht referenziert: Stattdessen wird das zeitliche Verhalten der abstrakten Maschine in imperativen Sprachen über den Zustand virtualisiert. In dieser Zeitvorstellung wird der Zustand über Daten wie z.B. die Werte von Variablen, Registerinhalte und die Stelle im Programm, die gerade ausgeführt wird, konstituiert. In Maschinencode wird diese Stelle durch den Programmzähler markiert. Ein Vorher und Nachher ist durch Zustandsübergänge gegeben. Physische Zeit wird erst in der Ausführung auf einer konkreten Maschine wieder relevant. So kann sich das physisch-zeitliche Verhalten eines semantisch gleichen Programms auf verschiedenen konkreten Maschinen auch sehr unterschiedlich darstellen. In imperativen Programmiersprachen finden sich dabei lineare Zeitvorstellungen wieder (die Reihenfolge von Instruktionen), jedoch kommen auch zyklische Vorstellungen (Schleifen) und sogar komplex verschachtelte Zyklen (Unterprogramme, Rekursionen) zum Einsatz. Demgegenüber stellen sich andere Programmierparadigmen, z.B. funktionale Sprachen, als vergleichsweise *zeitlos* dar. Rein funktionale Sprachen kommen völlig ohne Zustand aus. Analog zu mathematischen Funktionen werden Programme hier als Relationen betrachtet, Zeit spielt in dieser Konzeption zunächst keine Rolle. Erst in der Ausführung werden zeitliche Vorstellungen auf das Programm bezogen, auch ein ausgeführtes funktionales Programm benötigt z.B. Zeit für eine Berechnung. Dem Compiler kommt hierbei im Bezug auf die Zeit eine besondere Rolle zu: Er übersetzt das zustandslose funktionale Programm in Maschinencode, der das Maschinenverhalten zustandsbasiert (als virtuell zeitliche Abfolge von Instruktionen) beschreibt.

Eine besondere Herausforderung für Zeitvorstellungen in der Programmsemantik stellt Nebenläufigkeit dar. Werden im Quellcode mehrere gleichzeitig bearbeitbare Ausführungsstränge beschrieben, z.B. um durch Multithreading die Rechenkapazität von Mehrkernprozessoren auszunutzen oder um ein verteiltes System zu implementieren, ist die richtige Konzeption der Nebenläufigkeit, die unterschiedliches zeitliches Verhalten der einzelnen Ausführungsstränge korrekt behandelt, entscheidend. Die zwischen den einzelnen Ausführungssträngen geteilten Ressourcen und die Kommunikation der Stränge untereinander stellt insbesondere in Hinsicht auf zeitliches Verhalten einen eigenen Problembereich in der Informatik dar, der spezifische Techniken hervorgebracht hat, um mögliche Fallstricke der Nebenläufigkeit korrekt zu behandeln. Ein weiteres Gebiet, in dem Nebenläufigkeit und zeitliches Verhalten insgesamt innerhalb syntaktischer bzw. semantischer Programme eine große Rolle spielt, ist der gesamte Bereich des Hardwareentwurfs. Hardwarebeschreibungssprachen enthalten eigene Ausdrucksmöglichkeiten für zeitliches

Verhalten, um die massive Nebenläufigkeit von Schaltkreisen angemessen beschreiben zu können. Hier kommt eine zentrale auf die Zeit bezogene Technik zum Einsatz, die auf das ohne sie kaum beherrschbare zeitliche Verhalten großer Schaltkreise bezogen ist: Durch die selbstauferlegte Beschränkung der möglichen Schaltkreise auf synchrone Schaltkreise lassen sich deutlich komplexere Systeme konzipieren. Synchronität bezeichnet dabei die Eigenschaft, dass speichernde Elemente (z.B. Register) zu jeweils festen Zeiten über das gesamte System gleichzeitig aktualisiert werden. Diese festen Zeiten werden dabei durch ein Taktsignal vorgegeben, zu dem die z.B. aus taktgesteuerten Flipflops[174] bestehenden speichernden Elemente jeweils den am Eingang vorliegenden Wert speichern. In der Zeit zwischen den steuernden Taktflanken ändern sie ihren Zustand nicht. Diese Beschränkung zeitlichen Verhaltens ermöglicht den Umgang mit anderweitig deutlich komplexeren Schaltungen.

Eng mit den Zeitvorstellungen verknüpft sind Vorstellungen von Geschwindigkeit. Eine Gemeinsamkeit aller Bedeutungsdimensionen des Programmbegriffs ist dabei, dass Geschwindigkeit sich jeweils auf viele unterschiedliche Eigenschaften beziehen kann. In Geschwindigkeitszuschreibungen wird dabei eine andere Größe in Relation zu einer Zeitvorstellung gebracht. Die mechanische Geschwindigkeit physischer Gegenstände setzt beispielsweise eine zurückgelegte Strecke in Relation zur physischen Vorstellung von Zeit.

Auf die physischen Gegenstände in der Computertechnik bezogen kann Geschwindigkeit jedoch auch andere Größen in Relation zur physisch konstituierten Zeit setzen. Die Geschwindigkeit eines Datenträgers kann zum Beispiel die Datenrate, mit der sequenziell Daten geschrieben oder gelesen werden können, bezeichnen. Lese- und Schreibgeschwindigkeit können sich dabei unterscheiden, gleich ist der Bezug zur physischen Sekunde, die in der Angabe der die Raten verwendet wird. Allerdings können mit Geschwindigkeit auch Latenzzeiten gemeint sein (die z.B. durch die Positionierung des Schreib-Lesekopfes einer Festplatte bestimmt sind) oder die Anzahl an Lese- oder Schreiboperationen pro Sekunde, die der Datenträger durchführen kann. Jede dieser Größen kann in unterschiedlichen Anwendungszusammenhängen unterschiedlich stark ins Gewicht fallen. Schließlich kann mit Geschwindigkeit noch ein Mischwert dieser Größen gemeint sein, auch als Leistung oder *Performance* bezeichnet, der mit der physischen Zeit nur noch indirekt zusammenhängt. Ähnlich verhält es sich mit der Geschwindigkeit von Netzwerkverbindungen, mit der Durchsatz, *Round Trip Time* oder ein zusammengesetzter Wert mit weiteren Faktoren wie Paketverlusten bezeichnet werden kann.

Für die Ausführung von Programmen ist insbesondere die Geschwindigkeit von Prozessoren von Bedeutung. Als messbare Größe kann hier die Taktrate genannt werden. Diese bestimmt die Häufigkeit, mit der taktflankengesteuerte Flipflops im Prozessor beschrieben werden. Allerdings sagt der Wert alleine sehr wenig über die tatsächliche Prozessorleistung aus. Mit dieser sind etliche weitere Faktoren verknüpft, um nur einige zu nennen spielen die benötigten Takte für jede einzelne Instruktion, Vorhandensein und Tiefe einer Pipeline, Sprungvorhersage und spekulative Ausführung, Superskalarität, Anzahl und Art der Prozessorkerne, Zugriffszeiten und Größe von Caches und nicht zuletzt der Instruktionssatz eine große Rolle. Da das Zusammenspiel all dieser Faktoren einen analytischen Vergleich von Prozessorgeschwindigkeiten praktisch unmöglich macht werden

174 Die Nennung von Flipflops verweist auf eine zweite, von der Synchronität unterscheidbare, selbst auferlegte Beschränkung, die im Ergebnis deutlich komplexere Entwürfe zulässt: Die Beschränkung auf Schaltungen mit wertdiskreten Signalen. In diesen werden Signale wie z.B. Spannungen anstatt in ihrer Kontinuität als diskrete Werte aus einem beschränkten Bereich interpretiert. Im einfachsten (und am häufigsten angewendeten) Fall wird dieser Bereich binär gefasst, die Signale werden als 0 oder 1 interpretiert.

Mit der Synchronität zusammenhängend ist die Beschränkung auf zeitdiskrete Signale: Sie stellt eine Voraussetzung für synchrone Schaltungen dar. Die Beschränkung auf sowohl zeit- als auch wertdiskrete Signale (zusammen als Digitalsignal bezeichnet) ist Voraussetzung für die Handhabbarkeit der Komplexität z.B. von aktuellen Prozessoren.

für Vergleiche stattdessen Benchmark-Tests verwendet, die den Prozessor eine Reihe von Aufgaben berechnen lassen und dabei die Leistung messen. Bei den Messungen ist jeweils die physisch gedachte Zeit, die der Prozessor benötigt, ein wichtiges Kriterium – allerdings wird das Ergebnis der Benchmark-Tests aus vielen unterschiedlich gewichteten Einzelmessungen zusammengesetzt, häufig wird das Ergebnis als dimensionslose Kennzahl angegeben, der Bezug dieser Geschwindigkeitsbewertungen zur Zeit ist nur noch indirekt gegeben. Solche Benchmark-Tests testen dabei immer in den Zusammenhängen eines Gesamtsystems – die Ergebnisse können also auch von den anderen Komponenten abhängen.

Auf syntaktische und semantische Programme bezogen kann Geschwindigkeit je nach Anwendungsgebiet unterschiedliche Eigenschaften bezeichnen. Dies reicht von der Reaktionsgeschwindigkeit der Benutzeroberfläche über die Menge an Daten, die das Programm in einer bestimmten Zeit verarbeiten kann oder die Zeit, die für einen typischen Anwendungsfall benötigt wird bis zur Komplexitätsklasse des implementierten Algorithmus, die die benötigten Rechenschritte in Abhängigkeit zur Länge der Eingabe begrenzt. Wird die Geschwindigkeit von Programmen verglichen – beispielsweise wenn die neue Version eines Programms als langsamer oder schneller bezeichnet wird – wird dabei nicht unbedingt klar dargestellt, welches dieser Konzepte genau gemeint ist.

Wird Geschwindigkeit auf die Ausführung, also auf Programme als eingebettete Gegenstände, bezogen, so spielen hier sowohl die Eigenschaften der physischen Komponenten des Systems (wie Prozessor, Grafikkarte, Arbeits- und Langzeitspeicher usw.) eine Rolle als auch die maschinelle Semantik des Programms. Bei der Betrachtung von Geschwindigkeiten ist hierbei relevant, dass das Programm durch Compiler, Interpreter oder virtuelle Maschine in den Instruktionssatz des Prozessors übersetzt wird. Unterschiede im Instruktionssatz können dabei mehr oder weniger performante Übersetzungen ermöglichen. Dadurch können sich z.B. Erweiterungen des Instruktionssatzes in neuen Prozessorgenerationen als Geschwindigkeitssteigerungen darstellen – obwohl die Geschwindigkeit auf Instruktionsebene eigentlich gar nicht vergleichbar ist, da der für den neuen Prozessor generierte Maschinencode weder syntaktisch noch operationell semantisch auch nur ein gültiges Programm für den alten Prozessor darstellt. Erst auf einer höheren Abstraktionsebene (z.B. einer abstrakteren formalen Semantik oder einer informellen Beschreibung des Ein- und Ausgabeverhaltens) sind die Programme semantisch vergleichbar.

Die Formen von Zeit und Geschwindigkeit, die im Umfeld von Computerprogrammen konstitutiert werden, sind vielfältig. Einige dieser Konzepte stehen in einem engen Bezug zu physisch gedachter Zeit, andere sind damit nur lose verknüpft. Ein „schneller“ Algorithmus z.B. ist auf seiner Beschreibungsebene, der Semantik, nicht mit physischer Zeit verknüpft. Dieser Zusammenhang wird erst in der Ausführung, im Zusammenwirken mit den physischen Gegenständen des ausführenden Systems, hergestellt. In vielen Fällen findet dabei die Wahrnehmung zeitlicher Eigenschaften mehrfach vermittelt statt: Die Geschwindigkeitsmessung des Benchmark-Programms operationalisiert ein komplexes Konzept von Geschwindigkeit, führt darauf basierend Messungen durch und führt das Ergebnis in einigen wenigen Zahlen zusammen. Diese wiederum werden ausgegeben, z.B. als *Performance-Score* auf einem Monitor angezeigt. Erst durch die bei der Vermittlung durchgeführte Reduktion der vielen beteiligten Eigenschaften wird das Ergebnis als *Geschwindigkeit* des Systems erfahrbar und vergleichbar.

5. Fazit

5.1 Zusammenfassung

Was ist eigentlich ein Computerprogramm? In dieser Arbeit werden die vielfältigen Bedeutungen, in denen das Wort *Programm* auf Gegenstände bezogen wird, aufgegriffen und untersucht. Ausgehend davon wird ein Begriff entwickelt, der eine analytisch präzise Gegenstandsbestimmung möglich macht und gleichzeitig die Zusammenhänge in den Blick nimmt, in denen diese Gegenstände zueinander stehen. Von bestehenden ontologischen Untersuchungen, die Programme oder Software wesentlich charakterisieren und Fragen der Existenz und Identität von Programmen behandeln, unterscheidet sich das Vorgehen dieser Arbeit dabei in zwei grundlegenden Aspekten: Erstens wird als Ausgangspunkt für eine Begriffsbestimmung die menschliche Erfahrung herangezogen. Programme werden hier mithilfe derjenigen Eigenschaften untersucht, die ihnen in Wahrnehmungsprozessen zukommen, der entwickelte Begriff beschreibt sie als Erfahrungsgegenstand. Dieser Ausgangspunkt ermöglicht eine analytische Gliederung des Begriffs in mehrere klar unterscheidbare Bedeutungsdimensionen, die verschiedene Gegenstandsformen erfassen, in denen Computerprogramme wahrgenommen werden. Die Unterscheidung dieser Gegenstandsformen ermöglicht eine höhere Präzision bei der Begriffsverwendung. Zweitens wird diese Unterteilung als Ausgangspunkt verstanden, von dem ausgehend die Zusammenhänge, die zwischen diesen unterschiedlichen als Programm bezeichneten Gegenständen bestehen, untersucht werden können. Diese Zusammenhänge stellen den Grund dafür dar, dass es trotz der Vielfalt von Gegenständen, die unter den Programmbegriff fallen, sinnvoll für die Reflektion über Computertechnik ist, von einem einzelnen zusammenhängenden Begriff auszugehen. Die durch die Gliederung gewonnene analytische Präszision geht dadurch nicht verloren, da die einzelnen Bedeutungsdimensionen dieses zusammenhängenden Programmbegriffs jeweils benannt und ausführlich charakterisiert werden.

Als Erkenntniszugang zu diesem Verständnis von Computerprogrammen wird in dieser Arbeit auf die transzendentalphilosophischen Überlegungen in Immanuel Kants *Kritik der Reinen Vernunft* zurückgegriffen. Mithilfe dieser Überlegungen wird dargelegt, dass die Gegenstände, die unter den Programmbegriff fallen, zu einem großen Teil empirische Gegenstände sind. Sie werden mit menschlichen Sinnen wahrgenommen, Erkenntnis über sie ist synthetisch: Sie erfolgt durch ein Zusammenfügen von Erscheinungen und Begriffen. Lediglich ein bestimmter Teil der als Programme wahrgenommenen Gegenstände ist nicht empirisch: Ihre formale Semantik. Würden als *Programm* ausschließlich Gegenstände in dieser Bedeutung bezeichnet, so wäre Programm analog zu mathematischen Begriffen ein willkürlich gedachter Begriff – ein Begriff, der nicht von direkten empirischen Wahrnehmungen abhängt. Allerdings schließt ein solcher Begriff einen Großteil derjenigen Gegenstände aus, die in der Praxis als Computerprogramme verstanden werden und die für die Bedeutung, die Programmen in der alltäglichen Anwendung zukommt, maßgeblich sind. Aus diesem Grund ist der in dieser Arbeit entwickelte Programmbegriff ein empirischer – er erfasst das Computerprogramm als Erfahrungsgegenstand.

Als zweite Quelle des Erkenntniszugangs verwendet diese Arbeit die grundlegenden phänomenologischen Überlegungen Edmund Husserls. Die Phänomenologie fasst menschliche

Wahrnehmungen als intentionale Akte auf, der Gegenstand, auf den sie sich beziehen, wird durch die Wahrnehmung erst geformt, er wird im Wahrnehmungsakt konstituiert. Der Begriff der Gegenstandskonstitution wird in dieser Arbeit herangezogen, um die Programmwahrnehmung als Ereignis zu begreifen, die dem Gegenstand Programm erst eine bestimmte Form gibt. Computerprogramme werden dabei in wesentlich unterschiedlichen Formen konstituiert, so unterscheidet sich beispielsweise ein auf einem Computer laufendes Programm von seiner formalen Semantik. Anhand dieser unterschiedlichen Gegenstandsformen, in denen Computerprogramme konstituiert werden, werden die Bedeutungsdimensionen des hier vorgestellten Programmbegriffs entwickelt. Ein weiteres aus der Phänomenologie übernommenes Konzept ist Epoché, die Enthaltung gegenüber Vorannahmen bezüglich des Gegenstands. Dieses Konzept wird auf Computerprogramme in zwei wesentlichen Punkten bezogen: Erstens werden vorhandene Definitionen und Vorannahmen darüber, was Computerprogramme sind, nicht in der Begriffsbestimmung verwendet. Zweitens wird keine ontologische Bestimmung der Gegenstände unternommen. Überlegungen über ein erfahrungsunabhängiges Wesen von Programmen, z.B. die Identität zweier Programme oder die Dauer der erfahrungsunabhängigen Existenz betreffend, werden nicht vorgenommen.

Um die Zusammenhänge zwischen den einzelnen Bedeutungsdimensionen des entwickelten Programmbegriffs darzustellen wird der Begriff der *Appräsentation* verwendet. Dieser bezeichnet ein *mitgegenwärtig*-machen von Gegenständen in Wahrnehmungsakten. Ursprünglich wurde der Begriff von Edmund Husserl eingeführt, um die Wahrnehmung anderer Subjekte phänomenologisch zu erfassen. Alfred Schütz greift diesen Begriff auf, um die Bedeutung von Zeichen als appräsentiert zu erklären. In dieser Arbeit kennzeichnet Appräsentation den wahrgenommenen Zusammenhang zwischen in unterschiedlichen Gegenstandsformen konstituierten Computerprogrammen: Wird z.B. der Quellcode eines Programms gelesen, so kann sowohl die formale Semantik als auch eine spätere Anwendung des Programms mitgegenwärtig gemacht werden. Der Wahrnehmungsakt, der sich intentional auf den Quellcode bezieht, kann dabei einen Gegenstand mitgegenwärtig machen, auf den er sich nicht direkt bezieht. Im Beispiel ist das ausgeführte Programm bei der Wahrnehmung des Quellcodes appräsentiert, die Begriffsdimensionen des entwickelten Programmbegriffs werden bereits im Wahrnehmungsakt verknüpft.

Diese Verknüpfung besteht jedoch nicht nur auf der Ebene individueller Wahrnehmung, gleichzeitig ist die Verbindung von Quellcode mit einem laufenden Programm technisch und gesellschaftlich voraussetzungsreich. Im einfachsten Fall wird dazu mindestens eine Maschine benötigt, die das Programm ausführen kann, in der Praxis kommen Compiler, Linker, Assembler etc. dazu, die zwischen diesen unterschiedlichen Gegenstandsformen vermitteln. Um diese technische und gesellschaftliche Dimension zu erfassen greift diese Arbeit auf Bruno Latours Akteur-Netzwerk-Theorie zurück. Die Verbindung der verschiedenen Bedeutungsdimensionen wird dabei in Latours Terminologie als Verbindung heterogener Akteure, als Assoziation, verstanden.

Die phänomenologische Untersuchung der Gegenstandsformen, in denen Computerprogramme wahrgenommen werden, führt zu einer Unterscheidung von vier Begriffsdimensionen des entwickelten Programmbegriffs. Diese Dimensionen sind das Programm als räumlich-zeitlicher Gegenstand, das Programm als syntaktischer Gegenstand, das Programm als semantischer Gegenstand und das Programm als eingebetteter gegenstand. Diese Dimensionen lassen sich jeweils weiter untergliedern, um die Präzision bei der Zuordnung einzelner Gegenstände zu erhöhen. Als fünfte Begriffsdimension kommt noch die Untersuchung der anderen Dimensionen in ihren Zusammenhängen, den Assoziationen des

Programmbegriffs, hinzu.

Programme als räumlich-zeitliche Gegenstände sind dadurch gekennzeichnet, dass ihnen im Akt der Wahrnehmung zu einer bestimmten Zeit ein bestimmter Ort zugeschrieben werden kann. Die wichtigste Gegenstandsform, die unter diese Begriffsdimension fällt, sind physische Gegenstände. Diese umfassen insbesondere alle Datenträger, die Programme in physischer Form speichern (vom Blatt Papier bis zum SSD-Speicher), und physische Systeme, *auf denen* Programme ausgeführt werden. Die in der Gegenstandskonstitution zugeschriebenen Eigenschaften von Raum und Zeit fallen hierbei mit den physischen Vorstellungen von Raum und Zeit zusammen, physische Programme können mit den physischen menschlichen Sinnen wahrgenommen werden. Unter die räumlich-zeitlichen Programme fallen jedoch auch virtualisierte, nicht physisch gedachte Gegenstände. Dies sind beispielsweise Dateien, deren Verortung nicht notwendigerweise physisch konstituiert wird – sondern beispielsweise im virtuellen Raum eines Dateisystems oder eines Adressraums für URLs (Uniform Resource Locators).

Programme als syntaktische Gegenstände sind sprachliche Gegenstände, die nach bestimmten (syntaktischen) Regeln aufgebaut sind. In den häufig angewendeten Formen sind dies Ketten von Zeichen – als Quellcode, der nach den syntaktischen Regeln einer höheren Programmiersprache aufgebaut ist, aber auch als Maschinencode. Andere Formen wie z.B. Kontrollflussgrafen oder die grafischen Elemente visueller Programmiersprachen werden hier auch als syntaktische Programme verstanden – sie sind wie klassischer Quellcode nach bestimmten sprachlichen Regeln aufgebaut, auch wenn diese Regeln komplexere Formen als eindimensionale Zeichenketten zulassen. Die Zeichen werden im hier vorgestellten Begriff von ihrer Bedeutung, der Semantik, unterschieden.

Als semantische Gegenstände werden hier die Bedeutungen der syntaktischen Programme verstanden. Dabei wird zwischen zwei wesentlich unterschiedlichen Formen von Bedeutung differenziert: Die maschinelle Semantik eines Programms beschreibt ein abstraktes Maschinenverhalten. Diese kann als formale Semantik vorliegen, die mathematischen Methoden zugänglich ist. Allerdings werden auch die Bedeutungen der Programme unter Sprachspezifikationen, die keinem mathematischen Formalismus folgen, hier als maschinelle Semantik verstanden. Abstrakt ist die Semantik eines Programms, da sie ohne die Ausführung auf einer konkreten Maschine verstanden werden kann. Programme in höheren Programmiersprachen haben über diese Bedeutung hinaus häufig noch eine natürlichsprachliche Semantik, die zweite Form von Gegenstand, die hier als semantisches Programm verstanden wird. Über Schlüsselwörter, Bezeichner und Kommentare werden viele Elemente natürlicher Sprache im Quellcode eines Programms verwendet. Diese können unter anderem Erklärungen, Begründungen und Verknüpfungen zu spezifischem Wissen der Anwendungsdomäne enthalten. Da eine wichtige Eigenschaft des Quellcodes ist, dass er üblicherweise menschenlesbar und von Menschen bearbeitbar sein soll, wird die natürlichsprachliche Semantik hier als relevante Bedeutungsdimension der Erfahrung von Programmen aufgegriffen.

Die vierte Begriffsdimension umfasst alle Gegenstandskonstitutionen, in denen das Programm nur im Zusammenhang mit anderen Gegenständen begriffen, als in diese eingebettet verstanden werden kann. Eingebettete Programme umfassen dabei insbesondere alle Programme zu deren Laufzeit. Ein ausgeführtes Programm läuft auf einem komplexen System, das viele Gegenstände involviert, im einfachsten Fall liegt zumindest ein ausführender Prozessor vor. Hinzu kommen häufig flüchtige und langfristige Datenspeicher, weitere interne Komponenten von PCs oder Smartphones, Betriebssysteme und weitere Programme, Eingabe- und Ausgabegeräte, Sensoren und Aktoren bei eingebetteten Systemen usw. Das Programm

kann losgelöst von diesen anderen Gegenständen nicht ausgeführt werden, ohne einen Prozessor kann kein laufendes Programm wahrgenommen werden. Sie stellen daher das System dar, in das das laufende Programm eingebettet ist. Eine erweiterte Form von Einbettung umfasst die Funktionalität, Anwendungszusammenhänge und soziale, organisatorische Einbettung von Programmen. Diese Form von Einbettung beschreibt beispielsweise, wie ein Campus-Management-System in eine Universität eingebettet ist. Insbesondere ist das Programm hierbei auch in menschliches Handeln eingebettet.

Als Assoziation werden die verbindenden Konzepte verstanden, mit denen die vorhergehenden vier Bedeutungsdimensionen untereinander verknüpft werden. Hierbei wird herausgearbeitet, dass gegenwärtige Computertechnologie insgesamt und damit insbesondere Programme ohne die Herstellung gegenseitiger Verbindung zwischen den unterschiedlichen Gegenstandsformen gar nicht möglich wäre. So verknüpfen gesellschaftlich stabilisierte Programmiersprachen syntaktische und semantische Programme miteinander, aufwändig entwickelte und aktualisierte Compiler ermöglichen die Übersetzung in Maschinensprache, um eine Ausführung möglich zu machen. Weiter treten Computer selber vermittelnd auf, da sie die syntaktisch als Maschinencode geformten Programme in konkretes Maschinenverhalten übersetzen. In Bezug auf Bruno Latours Akteur-Netzwerk-Theorie wird dargelegt, dass hier viele Schritte der Übersetzung und Vermittlung zum Zuge kommen, und dass eine wesentliche Voraussetzung für die Funktionalität von Computern und Programmen die Stabilisierung dieser Vermittlungen in der Form von *Zwischengliedern* ist, welche Bedeutung ohne wesentliche Transformationen weitergeben können. Diese Stabilisierung von Zwischengliedern wird als soziotechnische Bedingung für die Programmkonstitution charakterisiert.

Im Anschluss wird der entwickelte Begriff auf drei Beispiele angewandt, um aufzuzeigen, wie die Berücksichtigung der unterschiedlichen Bedeutungsdimensionen und ihrer Zusammenhänge untereinander für Analysen hilfreich sein kann. Das erste Beispiel bezieht sich auf Programmtests und die Fehlerbehandlung in Programmen. Als zwei typische Tätigkeiten in der Softwareentwicklung werden diese Handlungen beispielhaft für alle die Herstellung von Programmen betreffenden Prozesse analysiert. Das zweite Beispiel wendet den Begriff auf formale Verifikation an. Diese besondere Form der Programmanalyse eignet sich, um Fragestellungen des Vertrauens in die Technik zu eröffnen. Die dritte Begriffsanwendung greift mit Open Source und Freier Software normative Bezüge zu Computerprogrammen auf.

Ergänzt wird die Begriffsentwicklung und anschließende Anwendung durch zwei Exkurse, die den entwickelten Programmbegriff in Bezug zu weiteren Konzepten setzen. Der erste Exkurs betrachtet das Programm als Komposition, hier werden Parallelen zur Komposition musikalischer Werke untersucht. Sinnbildlich kann dabei die Partitur als syntaktisches Programm, das Orchester als ausführendes System und die Aufführung als Ausführung verstanden werden. Programme wie musikalische Werke sind in dem Sinne komponiert, dass in ihnen die Dinge zusammen wirken. Der zweite Exkurs untersucht auf Computerprogramme bezogene Zeitvorstellungen. Dabei stellt sich heraus, dass sehr unterschiedliche Konzeptionen von Zeit eine Rolle spielen. Auch der eng mit der Zeit verknüpfte Begriff der Geschwindigkeit stellt sich auf Programme und Computer bezogen als vielschichtig dar, je nachdem auf welche Gegenstandsformen von Computerprogrammen er bezogen wird.

5.2 Ausblick

In dieser Arbeit wurde ein mehrdimensionaler Programmbegriff entwickelt, der die verschiedenen Formen, in denen Computerprogramme als Gegenstände menschlicher Erfahrung konstituiert werden, klar strukturiert und ihre wechselseitigen Zusammenhänge analysiert. Ein solcher Begriff stellt eine wesentliche Grundlage für eine Ästhetik von Computerprogrammen dar, die sich weitergehend mit der sinnlichen Wahrnehmung und den Zugängen, die der menschlichen Erfahrung zu Programmen möglich sind, beschäftigt. Der Begriff hat den Anspruch, sowohl theoretisch als auch empirisch anwendbar zu sein. Dadurch liegen Anwendungen in der Technikphilosophie, Techniksoziologie sowie der interdisziplinären Wissenschafts- und Technikforschung nahe.

In diesem Ausblick soll zunächst auf die Verallgemeinerbarkeit des Begriffs auf Programme, die gerade keine Computerprogramme sind, eingegangen werden. Anschließend werden drei größere Bereiche vorgestellt, die besonders geeignet für weitergehende Analysen mit Bezug zum hier entwickelten Programmbegriff erscheinen. Der erste Bereich ist die Vertiefung des Wahrnehmungszugangs und seine Anwendung auf spezifischere Fragestellungen, insbesondere im Hinblick auf aktuell diskutierte technische Entwicklungen wie z.B. die große Dynamik in der Digitalisierung, die massiven Fortschritte im maschinellen Lernen und die zunehmende Vernetzung im Internet der Dinge. Der zweite Bereich zielt, verknüpft mit der Ästhetik, auf ästhetische Urteile ab. Damit umfasst der Bereich alle Fragestellungen hinsichtlich der Wahrnehmung von Schönheit oder Eleganz von Programmen, allerdings auch die jeweiligen Gegenbegriffe: Wann ist ein Programm schön, wann ist es hässlich? Wann ist es sauber? Ist *quick and dirty* das Gegenteil von sauber? Wie verhält sich Eleganz zur Lesbarkeit? Der dritte Bereich umfasst ethische Fragestellungen und den Bezug gesellschaftlicher Normen auf Programme. Die möglichen Begriffsanwendungen reichen dabei von Untersuchungen darüber, wie ein Programm geschrieben werden soll (oder darf), was das Programm „tun“ darf, über die Betrachtung von Recht im Umfeld von Programmen bis zu einer Analyse von *Hackerethiken* und der spezifischen Programmauffassungen, die sich darin wiederfinden.

Der hier entwickelte Programmbegriff stellt eine Grundlage für weitere Untersuchungen zu Computertechnik, Prozessen der Digitalisierung und Veränderungen in der Technikgestaltung und -anwendung durch den verbreiteten Einsatz Programmgesteuerter Artefakte dar – insbesondere auch zu aktuellen Diskussionen über eine *vierte industrielle Revolution* und ihre Auswirkungen auf menschliche Arbeit soll der Begriff einen Beitrag leisten. Das Wort *Programm* ist jedoch älter als Computertechnik: Etymologisch geht es auf das griechische Wort πρόγραμμα zurück, mit dem (öffentliche) schriftliche Bekanntmachungen bezeichnet werden[175]. Neben den hier behandelten Computerprogrammen wird eine Vielzahl weiterer Gegenstände *Programm* genannt – einige Beispiele sind Abendprogramm, Forschungsprogramm, Parteiprogramm, Förderprogramm, das Programm eines Theaters, Fertigungsprogramm usw. Kann der entwickelte Programmbegriff auch zu Untersuchungen dieser Gegenstände beitragen? Anhand von zwei Beispielen soll die Möglichkeit einer Ausweitung des Begriffs gezeigt werden – die jedoch Gegenstand einer eigenständigen Arbeit sein müsste, da sie hier nicht abschließend behandelt

175 Siehe Kluge, Friedrich und Seebold, Elmar: Etymologisches Wörterbuch der deutschen Sprache. 25. Aufl. De Gruyter, 2011. S. 725.

und Kytzler, Bernhard, Redemund, Lutz und Eberl, Nikolaus: Unser tägliches Griechisch. 2. Auflage. Wissenschaftliche Buchgesellschaft, 2002. S. 856.

werden kann.

Mit Fernsehprogramm können analog zum Computerprogramm sehr unterschiedliche Gegenstände gemeint sein. Eine Programmzeitschrift wird räumlich-zeitlich als physischer Gegenstand konstituiert. Mit *Programm* kann jedoch auch der syntaktische Gegenstand der geschriebenen Zeichen darin gemeint sein – oder deren Bedeutung als semantischer Gegenstand. So kann ein Fernsehprogramm verlegt werden (physisch), als unübersichtlich empfunden werden (syntaktisch) oder der Inhalt bewertet werden (semantisch). Die Gemeinsamkeit der Programme, welche sie z.B. von anderen Zeitschriften oder Büchern unterscheidet, ist ihre Ausführbarkeit: Auch das Fernsehprogramm erlangt seine Bedeutung erst dadurch, dass das Verhalten anderer Gegenstände (und in diesem Fall auch Menschen) daran orientiert ist. Ein Fernsehsender ist als ausführendes System mit beteiligten Menschen zwar weniger leicht abzugrenzen als beispielsweise ein Smartphone als technisches System – allerdings steht auch hier die Erwartung im Vordergrund, dass das tatsächlich Gesendete dem semantischen Programm entspricht. Ein Unterschied besteht darin, dass Fernsehprogramme weniger stark formalisiert sind. Zwar enthält die Form, in der Sendungen und Zeiten in Tabellen aufgeführt werden, formalsprachliche Elemente – allerdings bestehen weitaus mehr Freiheiten in der Ausführung als bei Computerprogrammen.

Parteiprogramme (sowohl Grundsatzprogramme als auch Wahlprogramme) sind noch weniger formalisiert. Ihr Inhalt ist üblicherweise ausschließlich natürlichsprachlich. Auch hier können physisch, syntaktisch und semantisch konstituierte Gegenstände unterschieden werden. Auch Parteiprogramme können auf Papier oder in Dateien, auf Webseiten etc. verortet werden, auch hier können die Zeichen oder die Bedeutung gemeint sein. Die *Ausführbarkeit* dieser Programme stellt dabei gleichzeitig die wichtigste Gemeinsamkeit mit Computerprogrammen und den größten Unterschied zu ihnen dar. Die Gemeinsamkeit liegt darin, dass auch hier der Gegenstand erst durch die Möglichkeit einer Ausführung relevant wird – also durch die Erwartung, dass tatsächliche Politikgestaltung auf die Grundsatz- oder Wahlprogramme bezogen stattfindet. Analog zum Fernsehprogramm ist das System, in das das Programm eingebettet ist, komplex – es sind zahlreiche Menschen und weitere Gegenstände beteiligt. Der Unterschied zu Computerprogrammen liegt darin, dass bei der Politikgestaltung notwendigerweise eine bedeutungstransformierende Interpretation der Programme stattfindet – unter anderem Konkretisierungen, Gewichtungen, Anpassungen an aktuelle Gegebenheiten, Anpassungen durch demokratische Aushandlungsprozesse etc. Eine politische Partei, die sich an dem Programm orientiert, tritt nicht als Zwischenglied auf – sondern stets als Mittler, der Bedeutung zwar aufgreift und weitergibt, aber immer auch transformieren kann. Aus diesem Grund kann ein Parteiprogramm auch keine maschinelle Semantik haben – es lässt sich kein abstraktes System definieren (wie z.B. in Sprachspezifikationen), welches das Programm ausführt. Ein am Programm ausgerichtetes Verhalten ist hier immer konkret, da Konkretisierungen und weitere Interpretationen eine Anwendung erst möglich machen.

An den Beispielen wird deutlich, dass der entwickelte Programmbegriff auch über Computerprogramme hinaus zur Untersuchung von Programmen als Gegenstand menschlicher Erfahrung herangezogen werden kann. Die zentrale Gemeinsamkeit der verschiedenen Programme liegt in der Möglichkeit einer Ausführung: Das Verhalten eines Systems – bestehend aus weiteren Gegenständen und auch Menschen – wird unter anderem durch das Programm geformt. Wichtige Unterschiede liegen im Grad der Formalisierung des Programms und in den Freiheiten und Kontingenzen bei der Ausführung. Ein allgemeinerer Programmbegriff, der diese Unterschiede und Gemeinsamkeiten aufgreift, könnte in einer weiteren Arbeit entwickelt werden.

Ein wichtiger Ansatzpunkt zur Vertiefung der Ästhetik ist die Verknüpfung des Quellcodes mit den Anwendungen eines Programms. Hierbei wäre zu untersuchen, wie genau ein Teil der Wirklichkeit der Anwendungsdomäne im Programm abgebildet wird. Eine solche Analyse könnte sowohl den Abbildungsprozess in der Programmentwicklung als auch die Abbildung im Programmcode aufgreifen. Die Untersuchung der Entwicklungsprozesse könnte auf konkrete Beschreibungen aus der Softwaretechnik zurückgreifen und analysieren, wie die verschiedenen Techniken hier Wirklichkeit aus den Anwendungszusammenhängen abbilden. Dies umfasst die Modellierung, aber auch die Verwendung von Szenarien wie z.B. in Use Cases oder User Stories. Im Entwicklungsprozess selbst kann die Anwendung dabei auch durch Personen repräsentiert sein, in Scrum z.B. durch den *Product Owner* und die Stakeholder, mit denen dieser Rücksprache halten soll. Sie sollen dabei die konkreten Anforderungen an das Programm, die sich aus den Anwendungszusammenhängen ergeben, in den Entwicklungsprozess einbringen. Auf die Modellierung bezogen wäre zu untersuchen, wie genau Wirklichkeit beispielsweise in Diagrammen abgebildet wird, wann hier welche Information aufgegriffen wird und wie diese genau gedanklich mit den verschiedenen Einbettungen einer Anwendung in Verbindung gebracht werden.

Eine Untersuchung der Abbildung von Anwendungszusammenhängen im Programmcode selbst könnte die doppelte Abbildung von Domäneneigenschaften in der maschinellen Semantik des Programms und in natürlichsprachlichen Bedeutungen des Quellcodes aufgreifen. Zwei wichtige Aspekte, die hier untersucht werden könnten sind die Abbildung von Wissen über Zusammenhänge in der Anwendungsdomäne (maschinell z.B. in Datenstrukturen und deren Zusammenhängen repräsentiert – natürlichsprachlich z.B. in begründenden Kommentaren) und die Verwendung von sprachlichen Mitteln aus der Anwendungsdomäne. Eine solche Verwendung kann von domänenspezifischen Begriffen in Bezeichnern bis zur Implementierung einer maschinell verarbeitbaren domänenspezifischen Sprache gehen.

Eng mit diesen Fragestellungen verknüpft ist die Frage, ob und wie Programmiersprachen den Blick auf die Anwendungszusammenhänge mit prägen. Gibt es Spracheigenschaften, die eine Programmiersprache für bestimmte Anwendungsfelder mehr oder weniger geeignet machen? Insbesondere auf Programmierparadigmen bezogen könnte untersucht werden, inwiefern diese als Mittel zur Lösung von Aufgaben den Blick auf die Aufgaben selbst mitbestimmen. Untersuchungen zu diesen Fragestellungen könnten dazu beitragen, die Vielfalt verwendeter Programmiersprachen mit zu erklären: Ausgangspunkt ist hier, dass es sehr viele verschiedene höhere Programmiersprachen gibt. Fast jede davon ist Turing-vollständig, es kann also jede berechenbare Funktion darin ausgedrückt werden. Warum werden dann so viele unterschiedliche Sprachen verwendet? Ein Ansatz zur Beantwortung dieser Frage müsste vom menschlichen Zugang, der Eigenschaft der Lesbarkeit, ausgehen.

Schließlich könnte die Ästhetik noch im Hinblick auf spezifische Techniken vertieft werden. Dabei könnte zum Beispiel geklärt werden, wie das Verhältnis der verschiedenen Programmbedeutungen zueinander sich bei Anwendungen maschinellen Lernens darstellt. Hier spielt insbesondere das vom System (z.B. einem künstlichen neuronalen Netz) Erlernte eine entscheidende Rolle, die mit dem Programm verknüpften Daten (z.B. Verbindungsgewichtungen im künstlichen neuronalen Netz) verändern das maschinelle Verhalten grundlegender als dies üblicherweise bei „klassischen“ Programmen der Fall ist. Eine weitere Eigenschaft, die das trainierte künstliche neuronale Netz von einem klassischen Programm unterscheidet ist, dass das Verhalten des klassischen Programms mit Zugang zum Quellcode zumindest prinzipiell erklärbar ist (bei Maschinencode nur mit erheblichem

Aufwand). Wie ein künstliches neuronales Netz eine bestimmte Aufgabe löst oder warum es ein bestimmtes Verhalten zeigt ist dagegen nicht auf eine mit Quellcode vergleichbare Weise zugänglich[176].

Untersuchungen zu ästhetischen Urteilen über Programme könnten eine Vielzahl von Fragen über die Schönheit von Programmen oder über deren Abwesenheit behandeln. Ausgangspunkt dabei wäre die Frage, was es genau bedeutet, dass ein Computerprogramm schön ist. Strukturiert anhand der Bedeutungsdimensionen des hier entwickelten Programmbegriffs könnten zunächst die Schönheit des Programms in der Ausführung (z.B. die Eleganz und Zweckmäßigkeit einer Benutzeroberfläche, insbesondere auch alle für die Anwender_innen wahrnehmbaren Bilder und Töne) und die Schönheit physischer Gegenstände (das Design der eingebetteten Systeme, der Computer, der Datenträger, Dokumentation, Verpackung usw.) von auf den Quellcode bezogener Schönheit getrennt werden. Ästhetische Normen, die sich auf Eigenschaften des Quellcodes beziehen, können dahingehend untersucht werden, inwiefern sie auf die syntaktische Zeichenfolge, die Semantik oder auf das Verhältnis dieser beiden Gegenstandsformen abzielen. Beispielsweise betrifft ein Teil der Normen aus *Coding Conventions* das syntaktische Programm: Wie wird Whitespace verwendet, wie lang dürfen Zeilen sein, wo werden Einrückungen verwendet, in welcher Reihenfolge stehen verschiedene Programmelemente im Quellcode (wenn dies semantisch keinen Unterschied macht) usw. Ein anderer Teil der Normen bezieht sich auf die natürlichsprachliche Semantik von Programmen: Wie werden Variablen und Methoden benannt, wie werden Kommentare gestaltet, welche Informationen sollen sie vermitteln usw. Ein anderer Teil der Normen bezieht sich auf die maschinelle Semantik: Hierunter fallen z.B. Angaben, wie groß Klassen in objektorientierter Programmierung sein sollten, wie Datenstrukturen gestaltet werden, wann und welche *Design Patterns* verwendet werden, wie die jeweiligen Mittel der verwendeten Programmiersprache verwendet werden sollten.

Die ästhetischen Normen können auch in der Form von abstrakten Konzepten kommuniziert werden, Beispiele dafür wären Forderungen nach Einfachheit, Verständlichkeit, Modularität, Flexibilität, Übersichtlichkeit oder Modifizierbarkeit. Diese Konzepte können sich überschneiden und sie können sich jeweils sowohl auf das syntaktische Programm als auch auf seine beiden Semantiken beziehen. Ein wichtiger Schritt zum Verständnis ästhetischer Urteile über Programme wäre eine genauere Analyse dieser Konzepte unter Bezugnahme auf die Unterscheidung zwischen syntaktischem Programm, seiner maschinellen Semantik und seiner natürlichsprachlichen Semantik. Eine weitere relevante Frage in diesem Zusammenhang ist, inwiefern die Konzepte durch die verwendete Programmiersprache (insbesondere im Hinblick auf Programmierparadigmen), die Anwendungsdomäne und verwendete Softwareentwicklungsprozesse und Werkzeuge modifiziert werden.

Ausgehend hiervon könnten die Untersuchungen auf den Quellcode bezogen zu den Ausgangsfragen zurückkommen: Wann ist ein Programm *schön*? Wann ist es *hässlich*? Wann *elegant*? Kann eine *quick and dirty*-Lösung trotz der metaphorischen Widersprüchlichkeit sauber programmiert sein? Wann ist ein Programm *handwerklich gut gemacht*, wann ist es ein Produkt hoher *Programmierkunst*? Und wie hängen all diese ästhetischen Konzepte mit der Lesbarkeit von Quellcode zusammen? Diese Fragen können sowohl theoretisch untersucht werden als auch als Ausgangspunkt für empirische Forschung dienen, die die praktische

176 Lars-Erik Janlert nennt aus diesem Grund künstliche neuronale Netze, aber auch z.B. evolutionäre Algorithmen, Formen von *dark programming*. Techniken, die darunter Fallen, sind in ihrem Verhalten schwer nachvollziehbar, sie können allerdings im Gegensatz zu klassischen Programmen auch Aufgaben lösen, für die keine Vorstellung vorhanden ist, *wie* sie genau gelöst werden können.

Siehe Janlert, Lars-Erik: Dark programming and the case for the rationality of programs. In: Journal of Applied Logic Vol. 6 Iss. 4. 2008. S. 545-552.

Relevanz ästhetischer Urteile in der Entwicklungstätigkeit in den Blick nimmt. Andrew Oram und Greg Wilson haben Softwareentwickler_innen nach dem schönsten Stück Code gefragt, das sie kennen. Die im Buch *Beautiful Code* festgehaltenen Ergebnisse dieser Befragung geben hierfür einen Einstieg in *schönen* Code aus der Entwicklungspraxis. Yukihiro Matsumoto (der hauptverantwortliche Entwickler der Programmiersprache Ruby) vergleicht hier z.B. Programme mit Essays:

> „For essays, the most important question readers ask is, “What is it about?” For programs, the main question is, “What does it do?” In fact, the purpose should be sufficiently clear that neither question ever needs to be uttered. Still, for both essays and computer code, it’s always important to look at how each one is written. Even if the idea itself is good, it will be difficult to transmit to the desired audience if it is difficult to understand. The style in which they are written is just as important as their purpose. Both essays and lines of code are meant—before all else—to be read and understood by human beings. […]
>
> Computers can, of course, deal with complexity without complaint, but this is not the case for human beings. Unreadable code will reduce most people’s productivity significantly. On the other hand, easily understandable code will increase it. And we see beauty in such code.“[177]

Der dritte Bereich möglicher Untersuchungen, die auf dem hier entwickelten Programmbegriff aufbauen können, umfasst alle weiteren Fragestellungen bezüglich Normen, die sich auf Computerprogramme beziehen. Während der Bereich ästhetischer Urteile auch ästhetische Normen aufgreift geht es hier insbesondere um die Frage, was ein Programm „tun darf“ oder „tun soll“. Auf menschliche Akteure reduziert würde dadurch untersucht, was programmiert werden darf oder soll. Hierbei kann der Begriff sowohl in den ethischen Überlegungen als auch in der Untersuchung gesellschaftlicher Normen angewendet werden.

In ethischen Überlegungen zu Computerprogrammen ist es naheliegend, zunächst auf die Ausführung einzugehen, auf das eingebettete Programm. Hier beziehen sich Normen beispielsweise darauf, welche Daten erhoben werden dürfen, wie sie verarbeitet werden sollen und wann sie gelöscht werden müssen. Bei eingebetteten Systemen wird diskutiert, wie sich Maschinen verhalten dürfen: Wie soll ein autonomes Fahrzeug in bestimmten Situationen reagieren, wie darf die menschliche Arbeit in der *Industrie 4.0* durch programmgesteuerte Maschinen modifiziert werden, welche Grenzen werden dem „Handeln“ künstlicher Intelligenzen gesetzt? Hierbei wird stets die Einbettung des ausgeführten Programms mit der maschinellen Semantik verknüpft: Die Semantik soll so gestaltet sein, dass das Programm in der Ausführung den normativen Ansprüchen genügt.

Allerdings wurde im Anwendungskapitel zu Open Source und Freier Software bereits aufgezeigt, dass sich Normen auch auf die weiteren Bedeutungsdimensionen des Programmbegriffs erstrecken können. So schreiben die Lizenzen Freier Software fest, dass der Quellcode eines Programms physisch verfügbar sein muss – sei es als Download oder als auf Anfrage versendeter Datenträger. Weiter betreffen die Normen Freier Software nicht nur die Semantik – diese soll gerade auch syntaktisch in einer menschenlesbaren Form erhältlich sein. Das Programm wird hier nur dann als *vertrauenswürdig* aufgefasst, wenn die maschinelle Semantik menschlicher Erfahrung für jeden zugänglich ist – diese Lesbarkeit umfasst auch die natürlichsprachliche Semantik.

177 Matsumoto, Yukihiro: Treating Code as an Essay. In: Oram, Andrew, and Greg Wilson: Beautiful code. O'reilly, 2007. S. 477f. (aus dem Japanischen übersetzt von Nevin Thompson)

Reicht nun die Verfügbarkeit und Lesbarkeit des Quellcodes aus, um Programmen zu vertrauen? Die Herausforderungen einer Beantwortung dieser Frage liegen unter anderem darin begründet, dass zu manchen Programmen gar kein lesbarer Quellcode existiert[178] - insbesondere aber würde diese Fassung von Vertrauen Programmierkenntnisse voraussetzen. Reicht es also aus, dass *andere* Anwender_innen den Quellcode lesen können und mögliche Vertrauensbrüche finden würden? Dies würde voraussetzen, dass Quellcode nicht „lügen" kann. Allerdings gibt es auch Quellcode, der genau das versucht: *Underhanded Code*[179]. Dieser versteckt „bösartige" Funktionalität in scheinbar harmlosem Quellcode.

Der Ausblick konnte einige Gebiete weiterer aussichtsreicher Forschung umreißen. Der in dieser Arbeit entwickelte Programmbegriff stellt für diese Untersuchungen ein hilfreiches Werkzeug zur Verfügung: Die verschiedenen Formen, in denen Computerprogramme als Gegenstand konstituiert werden, können durch ihn präzise benannt und klar voneinander unterschieden werden. Gleichzeitig greift der Begriff die Zusammenhänge zwischen diesen Gegenstandsformen auf und macht sie einer Analyse zugänglich. Durch den gewählten Ausgangspunkt zur Begriffsentwicklung wird das Computerprogramm dabei so erfasst, wie es sich der menschlichen Wahrnehmung darstellt: als Erfahrungsgegenstand.

178 Siehe Fußnote 176 zu *dark programming*.

179 Diese Bezeichnung findet in Anlehnung an den bekannten *Underhanded C Contest* statt, siehe http://www.underhanded-c.org/ (abgerufen am 03.10.2018).

6. Anhang

6.1 Literaturverzeichnis

Adam, Barbara: Das Diktat der Uhr. Suhrkamp, 2005.

Benacerraf, Paul: Mathematical Truth. In: The Journal of Philosophy, Vol. 70, No. 19. 1973. S. 661-679.

Berger, Peter L., Luckmann, Thomas: The Social Construction of Reality. Penguin Books, 1966.

Bijker, Wiebe E.: Of bicycles, bakelites, and bulbs: toward a theory of sociotechnical change. MIT Press, 1995.

Bracha, Gilad: Preface to the Third Edition. In: Gosling, James et al: The Java Language Specification. Third Edition. Addison-Wesley, 2005. S. xxxi-xxxiv.

Brooks, Frederick P.: The Mythical Man-Month. Essays on Software Engineering. Anniversary Edition. Reading, Addison-Wesley. 1995 (1975).

Carnap, R.: Formalwissenschaft und Realwissenschaft. In: Erkenntnis, Vol. 5 Issue 1. 1935.

Chung, Tae-Sun, et al.: A survey of flash translation layer. In: Journal of Systems Architecture Vol. 55 Issue 5-6. 2009. S. 332-343.

Colburn, Timothy, Shute, Gary: Abstraction in computer science. Minds and Machines Vol. 17 Issue 2. 2007. S, 169-184.

Daylight, Edgar G.: Turing Tales. Lonely Scholar, 2016.

Drossopoulou, Sophia, und Eisenbach, Susan: Describing the semantics of Java and proving type soundness. In: Alves-Foss, Jim: Formal Syntax and Semantics of Java. Springer, 1999. S. 41-80.

Gosling, James et al.: The Java Language Specification. Third Edition. Addison-Wesley, 2005.

Grassmuck, Volker: Freie Software. Zwischen Privat- und Gemeineigentum. Bonn. Bundeszentrale für politische Bildung, 2002.

Harman, Mark: Why Source Code Analysis and Manipulation Will Always Be Important. In 10th IEEE Working Conference on Source Code Analysis and Manipulation (SCAM). 2010. S. 7-19.

Harris, David M., Harris, Sarah L.: Digital design and computer architecture. 2nd Ed. Morgan Kaufmann, 2013.

Hennessy, John L., und Patterson, David A.: Computer architecture: a quantitative approach. 5th Ed. Elsevier, 2011.

Hunt, Andrew and Thomas, David: The Pragmatic Programmer. From Journeyman to Master. Addison-Wesley. 1999.

Husserl, Edmund: Husserliana 6, Die Krisis der Europaischen Wissenschafien und die Transzendentale Phänomenologie, herausgegeben von Walter Biemer. Nijhoff, 1976.

Husserl, Edmund: Husserliana 1, Cartesianische Meditationen und Pariser Vorträge, Hrsg. Und eingeleitet von Stephan Strasser. Nijhoff, 1950.

Husserl, Edmund: Ideen zu einer reinen Phänomenologie und phänomenologischen Philosophie. Niemeyer, 1913.

Hyland, J. M. E. und Ong, C.-H. L: On Full Abstraction for PCF: I, II, and III. In: Information and Computation, vol. 163, no. 2. 2000. S. 285-408.

Ihde, Don: Postphenomenology: Essays in the Postmodern Context. Northwestern University Press, 1993.

Ihde, Don: Technology and the Lifeworld: From Garden to Earth. Indiana University Press, 1990.

Irmak, Nurbay: Software is an abstract artifact. Grazer Philosophische Studien Vol. 86 Issue 1. 2012. S. 55-72.

Janlert, Lars-Erik: Dark programming and the case for the rationality of programs. In: Journal of Applied Logic Vol. 6 Iss. 4. 2008. S. 545-552.

Kant, Immanuel: Kritik der reinen Vernunft. Suhrkamp, 1974. Erstauflage 1781.

Kernighan, Brian W., and Phillip James Plauger: The elements of programming style. 2nd Ed. McGraw-Hill, 1978.

Kluge, Friedrich und Seebold, Elmar: Etymologisches Wörterbuch der deutschen Sprache. 25. Aufl. De Gruyter, 2011.

Knuth, Donald: The Art of Computer Programming. Vol. 1 / Fundamental Algorithms. Addison-Wesley. 1968.

Kripke, Saul A.: Wittgenstein über Regeln und Privatsprache. Suhrkamp, 1987. Englische Erstausgabe 1982.

Kytzler, Bernhard, Redemund, Lutz und Eberl, Nikolaus: Unser tägliches Griechisch. 2. Auflage. Wissenschaftliche Buchgesellschaft, 2002.

Latour, Bruno: Eine neue Soziologie für eine neue Gesellschaft. Einführung in die Akteur-Netzwerk-Theorie. Aus dem Englischen von Gustav Roßler. Suhrkamp, 2007.

Long, Fred, et al.: Java coding guidelines: 75 recommendations for reliable and secure programs. Addison-Wesley. 2013.

Mancosu, P., Zach, R., & Badesa, C.: The development of mathematical logic from Russell to Tarski, 1900-1935. 2008. in: Haaparanta, L: The development of modern logic. Oxford University Press, 2009. S. 318-470.

Matsumoto, Yukihiro: Treating Code as an Essay. In: Oram, Andrew, and Greg Wilson: Beautiful code. O'reilly, 2007. S. 477-481. (aus dem Japanischen übersetzt von Nevin Thompson)

Myers, Glenford J., Sandler, Corey, Badgett, Tom: The Art of Software Testing. Third Edition. Wiley, 2012.

Nielson, Hanne Riis, und Nielson, Flemming: Semantics with applications. Springer, 2007 (1992).

Nordmann, Alfred: Object lessons: towards an epistemology of technoscience. In: Scientiae

Studia Vol. 10. 2012. S. 11-31.

Nordmann, Alfred: Philosophy of technoscience in the regime of vigilance. In: Hodge, Graeme A., Bowman, Diana M., Maynard, Andrew D.: International handbook on regulating nanotechnologies. Edward Elgar Publishing, 2010. S. 25-45.

Nowotny, Helga: Eigenzeit. Entstehung und Strukturierung eines Zeitgefühls. Suhrkamp. 1995 (Erstauflage 1993).

Oram, Andrew, and Greg Wilson: Beautiful code. O'Reilly, 2007.

Page, Daniel: A Practical Introduction to Computer Architecture. Springer, 2009.

Pinch, Trevor J., Bijker, Wiebe: The Social Construction of Facts and Artifacts: Or How the Sociology of Science and the Sociology of Technology Might Benefit Each Other. In: Bijker, Wiebe E. Et al.: The social constructions of technological systems: New directions in the Sociology and History of Technology. MIT Press, 1993 (1987). S. 17-50.

Raymond, Eric S,: The Cathedral and the Bazaar. Musings on Linux and Open Source by an Accidental Revolutionary. O'Reilly, 2001.

Rosenberger, Robert, Verbeek, Peter-Paul: A field guide to postphenomenology. In: Rosenberger, Robert, Verbeek, Peter-Paul: Postphenomenological investigations: Essays on human-technology relations. Lexington, 2015. S. 9-42.

Russell, Bertrand, et al.: Philosophie des Abendlandes: ihr Zusammenhang mit der politischen und der sozialen Entwicklung. Europaverlag, 1950.

Schütz, Alfred: Der sinnhafte Aufbau der sozialen Welt. Eine Einleitung in die verstehende Soziologie. UVK, 2004.

Schütz, Alfred: Theorie der Lebenswelt 2. Die kommunikative Ordnung der Lebenswelt. UVK, 2003.

Schütz, Alfred, Luckmann, Thomas: Strukturen der Lebenswelt. Band 1. Suhrkamp, 1979.

Schroeder, B., Lagisetty, R., Merchant, A.: Flash Reliability in Production: The Expected and the Unexpected. In: Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16). 2016. S. 67-80.

Sipser, Michael: Introduction to the Theory of Computation. Third Edition. Cengage Learning, 2012.

Slonneger, Kenneth, Kurtz, Barry L.: Formal Syntax and Semantics of Programming Languages. Reading. Addison-Wesley. 1995.

Soare, Robert I.: Turing Oracle Machines, Online Computing, and Three Displacements in Computability Theory. In: Annals of Pure and Applied Logic Vol. 160 Issue 3. 2009. S. 368-399.

Sommerville, Ian: Software Engineering. Ninth Edition. Pearson, 2011.

Tanenbaum, Andrew S., Bos, Herbert: Modern Operating Systems. Fourth Edition. Pearson, 2015.

Tanenbaum, Andrew S., Wetherall, David J.: Computer Networks. Fifth Edition. Prentice Hall, 2010.

Turner, Raymond: Programming Languages as Technical Artifacts. In: Philosophy & Technology Vol. 27 Iss. 3. 2014. S. 377-397.

Turner, Raymond: Programming Languages as Mathematical Theories. In: Vallverdú, Jordi: Thinking Machines and the Philosophy of Computer Science: Concepts and Principles. 2010. S. 66-82.

Turner, Raymond: Understanding programming languages. In: Minds and Machines Vol. 17 Iss. 2. 2007. S. 203-216.

Weber, Max: Wirtschaft und Gesellschaft. Mohr Siebeck, 1922.

Weber, Max: Die rationalen und soziologischen Grundlagen der Musik. Drei Masken Verlag, 1921.

Wittgenstein, Ludwig: Philosophische Untersuchungen. 6. Auflage. Suhrkamp , 2013 (2003). Erstausgabe 1953.

6.2 Referenzierte Webseiten

Freie Software, GNU-Projekt:

https://www.gnu.org/philosophy/free-sw.html (abgerufen am 18.09.2018).

https://www.fsf.org/about/what-is-free-software (abgerufen am 18.09.2018).

https://www.gnu.org/philosophy/philosophy.html (abgerufen am 18.09.2018).

https://www.gnu.org/licenses/license-list.html#SoftwareLicenses (abgerufen am 18.09.2018).

https://www.gnu.org/philosophy/open-source-misses-the-point.en.html (abgerufen am 18.09.2018).

Open Source:

https://opensource.org/about (abgerufen am 18.09.2018).

https://opensource.org/licenses/category (abgerufen am 18.09.2018).

Ubuntu:

https://www.ubuntu.com/about (abgerufen am 21.08.2018).

https://wiki.ubuntu.com/UbuntuFlavors (abgerufen am 21.08.2018).

Weitere Webseiten:

https://www.iso.org/obp/ui/#iso:std:iso-iec:2382:ed-1:v1:en (abgerufen am 29.06.2018).

https://www.tu-darmstadt.de/ (abgerufen am 26.6.2018).

https://www.unicode.org/standard/WhatIsUnicode.html (abgerufen am 31.07.2018).

http://www.oracle.com/technetwork/java/codeconventions-150003.pdf (abgerufen am 05.08.2018).

http://lists.uclibc.org/pipermail/uclibc/2014-July/048400.html (abgerufen am 20.10.2018).

http://www.underhanded-c.org/ (abgerufen am 03.10.2018).

https://creativecommons.org/licenses/ (abgerufen am 24.02.2019).