MATTHEW FULLER / ANDREW GOFFEY

DIE OBSKUREN OBJEKTE DER OBJEKTORIENTIERUNG

Einleitung

Objektorientierung bezeichnet einen Ansatz der Elektronischen Datenverarbeitung (EDV), bei dem Programme als Wechselwirkungen zwischen programmatisch definierten Objekten – rechnerischen Objekten – angesehen werden, anstatt als sequenzielle Ausführung von Aufgaben, die durch eine Liste von Befehlen, Routinen oder Subroutinen definiert sind. Objekte im Sinne der objektorientierten Programmierung sind Zusammenschlüsse von Daten sowie der Methoden, die auf diesen Daten ausgeführt werden können, bzw. zustandsorientierte Abstraktionen *(engl.: stateful abstractions)*. Im Kalkül der Objektorientierung kann alles ein rechnerisches Objekt sein und alles was berechnet werden kann, muss wiederum ein Objekt bzw. eine Eigenschaft eines Objekts sein. Objektorientierte Programmierung unterscheidet sich dabei sowohl von früheren prozeduralen oder funktionalen Programmieransätzen (verkörpert durch Sprachen wie C und Lisp) als auch von der deklarativen Programmierung (Prolog) und jüngst der komponentenbasierten Programmierung. Einige der heute am häufigsten benutzten Programmiersprachen – Java, Ruby, C# – haben eine betont objektorientierte Ausrichtung oder wurden explizit objektorientiert konzipiert. Und obwohl dieser Ansatz von einigen kritisiert wird, ist er ebenso tief im Denken vieler Informatiker und Softwareentwickler verankert wie in den multiplen digital-materiellen Schichten *(strata)* gegenwärtiger sozialer Beziehungen.

Dieser Aufsatz untersucht einige Aspekte des *turns* zu Objekten in der Welt der Computerprogrammierung (eine generische Bezeichnung, die Elemente der Informatik und der Softwareentwicklung umfasst). Er fragt, welche Macht rechnerischen Objekten zugeschrieben wird, welche Effekte dies mit sich bringt und – vielleicht noch wichtiger – wie diese Effekte entstehen. Um die technische Welt der Computerprogrammierung im breiteren Kontext der sich ändernden Machtstrukturen in gegenwärtigen Gesellschaften zu situieren, schlagen wir

vor, Programmierung als rekursive Problemlösung aus und mit digitalen Materialien zu verstehen: Dabei werden Beziehungen, die auf verschiedenen Realitätsstufen wirksam sind, verdichtet und abstrahiert, es entstehen neue Formen von Handlungsmacht und der Formalismus algorithmischer Operationen bestimmt die Maßstäbe, auf denen digitale Materialien wirksam und erfinderisch werden. Dieser Aufsatz versucht also das, was man die Territorialisierungsmacht rechnerischer Objekte nennen könnte (eine Menge von Einflüssen, die Beziehungen, Prozesse und Praktiken in einem paradoxerweise abstrakt-materiellen Raum modellieren und remodellieren), wahrnehmbar zu machen.[1]

Auf der einen Seite wird das Rechnen in breiter und vielfältiger Weise als Metapher für kognitive Prozesse eingesetzt (man beachte die anhaltende Suche nach Künstlicher Intelligenz), auf der anderen Seite als Synekdoche einer mechanisierten, entmenschlichten und entfremdeten Industriegesellschaft erfahren – beides Kehrseiten der gleichen epistemischen Münze. Als solches könnte Rechnen als irgendwie losgelöst von der reichhaltigen materiellen Textur der Kultur und von der Beschäftigung mit der ontologischen Dimension von ‹Dingen› erscheinen. In der Tat mag die EDV, mit ihrem konzeptuellen Hintergrund in der von David Hilbert ausgelösten formalistischen Revolution der Mathematik, nicht dazu auserkoren sein, uns besonders viel über die Natur der Dinge oder Objekte überhaupt zu sagen. In der präzisen Ausprägung ihres allgemeinen Anspruchs auf Universalität (soweit man ‹im Allgemeinen› über Formalismus sprechen kann) für alle Objekte valide zu sein (eigentlich sogar für jede beliebige Entität) ist Formalismus per Definition ohne jegliches reales Objekt und bietet stattdessen eine symbolische Aussicht auf das, was, um sein zu können, von der Art her eine Variable sein muss. Objekte und Dinge – in der Breite ihrer materiellen Texturen und ihrer Faktizität – neigen dazu, sich in das formale Kalkül von Zeichen zu verwandeln.

Die Details der transformativen Operationen, die durch Programmierung erreicht werden sollen – immer etwas raffinierter als die einfache ‹Symbolmanipulation› –, haben bisher ebenso wenig Aufmerksamkeit erfahren wie die Handlungsmacht von rechnerischen Objekten selbst in diesen Transformationen. In diesem Aufsatz greifen wir diese Frage durch eine kurze Betrachtung der objektorientierten Programmierung und ihrer transformativen Auswirkungen auf. Wir betrachten den rechnerischen Formalismus mithilfe der Techniken und Technologien der Informatik und der Softwareentwicklung, um Objektorientierung als eine soziotechnische Praxis zu behandeln.

Als eine solche Praxis hat sie mehr als nur eine flüchtige Ähnlichkeit zu verschiedenen Mitteln, um Experimentalobjekte und Prozesse zu disziplinieren, wie sie von Andrew Pickering beschrieben wurden. Die wirkungsvolle Lösung eines Rechenproblems ist eine Frage der erfolgreichen Erstellung einer mehr oder weniger stabilen Menge materieller Prozesse innerhalb, aber auch außerhalb der Haut der Maschine durch Programmierung.[2]

1 Wir entnehmen das Konzept der Territorialisierung und die verwandten Begriffe – De- und Re-Territorialisierung – von Gilles Deleuze und Felix Guattari. Sie werden weiter unten ausführlich behandelt.

2 Andrew Pickering, *The Mangle of Practice: Time, Agency and Science*, Chicago (Chicago Univ. Press) 1995.

Sprachen der Objekte und Ereignisse

Um die transformativen Kapazitäten zu verstehen, die rechnerische Objekte mit sich bringen, muss man zunächst die Entwicklung der Programmiersprachen betrachten, da mit der Erfindung der Programmiersprachen allererst die umfassenden Parameter festgelegt wurden, mithilfe derer eine Maschine mit der Außenwelt – mit sich selbst, mit anderen Maschinen, mit dem Menschen, mit der Umwelt usw. – kommunizieren kann.

Programmiersprachen – vermittelnde Grammatiken zum Schreiben eines Sets von Befehlen (Algorithmen und Datenstrukturen), die in ausführbaren Maschinencode übersetzt bzw. kompiliert werden – sind anders als die im Gegensatz dazu als ‹natürlich› bezeichneten Sprachen. Sie sind anders nicht nur in dem Sinne, dass sie andere Grammatiken besitzen, sondern auch dadurch, dass sie in einem spezifischen Kontext entworfen worden sind, der auf ein spezielles Set soziotechnischer Anordnungen, eine Konstellation von Kräften – Maschinen, Techniken, institutionelle und wirtschaftliche Vorkehrungen – fokussiert ist. Eine Programmiersprache ist ein sorgfältig und präzise konstruiertes Set von Protokollen, die in Bezug auf historisch, technisch, organisatorisch usw. genau bestimmte Probleme eingeführt worden ist. In der Regel mit einer Vielzahl expliziter – oft technischer, manchmal aber auch ästhetischer – Erwägungen konzipiert ist Programmiersprachen dennoch die spezifische Konfiguration von Gefügen (engl: *assemblages*) eingeschrieben, aus denen sie hervorgegangen sind, sowie die Anforderungen und Zwänge, die diese erzeugt haben – gerade so, wie sie selbst die Erzeugung neuer Gefüge ermöglichen. Der Informatiker, könnte man sagen, «invents assemblages starting from assemblages which have invented him [sic] in turn».[3]

Das Vorhaben der Objektorientierung in der Programmierung entsteht erstmals mit der Entwicklung der Sprache SIMULA. SIMULA wurde von Kristen Nygaard und Ole-Johan Dahl Anfang der 1960er Jahre am Norwegian Computing Centre entwickelt.[4] Wie der Name andeutet, ging es darum, Mittel zur Beschreibung – d. h. zur Programmierung – von Arbeitsabläufen *(workflows)* bereitzustellen und diese zu simulieren. Das Ziel einer solchen Simulation war, die Fähigkeit zum Entwerfen von Arbeitssystemen – trotz ihrer technisch recht hohen Komplexität – in den Wirkungsbereich derer zu bringen, die Arbeitsstätten *(workplace)* einrichteten. Als solches hatte das Projekt vieles mit anderen Entwicklungen höherer Programmiersprachen und Datenbanksystemen jener Zeit gemein, die darauf abzielten, technische Prozesse an die Denkweise von Nicht-Technikern anzunähern.[5] Gleichermaßen waren sie ein Weg, um die formale Beschreibung der Welt unter der Haut des Computers hervorzuholen. SIMULA versuchte, das Expertenwissen des Programmierers in Einklang mit Entscheidungssystemen einer Arbeitswelt zu bringen, in der eine sozialdemokratische Version der Arbeiterräte[6] den Entwurf von Arbeitsabläufen bestimmte. Diese Tendenz entsprang dem später als partizipatorisches Design bekannt

3 Gilles Deleuze, Claire Parnet, *Dialogues*, London (Athlone) 1987, 52. Übersetzung leicht modifiziert.

4 Kristen Nygaard, Ole-Johan Dahl, The Development of the SIMULA Languages, in: *ACM SIGPLAN Notices*, Fol. 13, No. 8, Aug. 1978. 245–272. Jaroslav Sklenar, Introduction to OOP in SIMULA, in: *30 Years of Object Oriented Programming*, Univ. of Malta, 5.12.1997, online unter: http://staff.um.edu.mt/jskl1/talk.html, gesehen am 15.2.2012; Jan Rune Holmevik, Compiling SIMULA: A Historical Study of Technological Genesis, in: IEEE *Annals of the History of Computing*, Vol. 16, no. 4, 1994, 25–37.

5 Wie z. B. in den Arbeiten von Edgar Codd.

6 Eine ausführliche Beschreibung der Arbeiterräte ist hier nicht möglich. Eine historische Präzedenz und ein fortgeschrittener Arbeitsstrang finden sich jedoch bei Jacques Cammatte und Anton Pannakoek.

7 Operations Research bzw. Operational Research in den USA ist

gewordenen Prinzip, wurzelt aber zugleich in einer Version der Unternehmensforschung *(operations research)*,[7] in der Arbeitsanalysen im Hinblick auf die Verminderung anstrengender Arbeit durchgeführt wurden.[8]

SIMULA 1, die erste Version von SIMULA, wurde zunächst nicht entwickelt, um Objektorientierung als ein neues Format für Programmiersprachen zu etablieren, sondern vielmehr als eine effektive Möglichkeit, die Transaktionen komplexer Systeme zu modellieren. Eine Simulation von Arbeitsprozessen wird in dem Moment wünschenswert, in dem Systeme einen Grad an Komplexität erreicht haben, der ihr Verstehen zu einer nicht-trivialen Aufgabe macht. In SIMULA wurde diese Aufgabe zunächst als eine Serie von «discrete event networks»[9] aufgefasst, in denen inventarische, einreihende, verarbeitende und materielle Prozesse modelliert werden können und in denen es eine klare Korrelation zwischen den Eigenschaften des Arbeitsprozesses und der Art gibt, wie dieser als Gebilde von rechnerischen Objekten modelliert wurde. Die implizite Ontologie von *discrete event networks* (das *‹network›* fiel später weg), die von SIMULA genutzt wurde, um solche epistemisch adäquaten Modelle von komplexen Prozessen zu erschaffen, war eine, in der realweltliche Prozesse in Begriffen von Ereignissen – oder Aktionen – verstanden wurden, die zwischen Entitäten stattfinden, und nicht etwa als beständiges Set von Beziehungen zwischen ihnen. Die Sprache selbst sollte dabei den Forscher zwingen, darauf zu achten, dass alle Aspekte des Prozesses berücksichtigt werden und diese Aufmerksamkeit in die angemessenen Bahnen lenken.

Anfangs kam der Sprache nur wenig Bedeutung in der allgemeinen Programmierung zu. Tatsächlich wurde erst mit SIMULA 67 das wichtigste technische Merkmal für objektorientierte Programmierung skizziert – nämlich die Eigenschaft, Programme zu schreiben, die Daten und Prozesse zu einer dauerhaften Entität vereinen[10] –, was durch Spezifizierung von ‹Klassen› und ‹Unterklassen› erreicht wurde, die beide in der Lage sind, bestimmte Aktionen zu starten oder auszuführen. SIMULA selbst wurde nicht als vollwertige Programmiersprache eingesetzt,[11] da die vorhandenen Rechenkapazitäten noch nicht ausreichten, um die Sprache effektiv einzusetzen. In diesem frühen Stadium der Geschichte der Programmierung und der Programmiersprachen stand die Möglichkeit einer Sprache, welche so wesentlich dafür angepasst war, realweltliche Prozesse zu beschreiben, zwangsläufig im Widerspruch zu den praktischen Hindernissen, effiziente Programme zu schreiben.

Dennoch wurden SIMULAs Neuerungen – strukturierte Codeabschnitte, genannt: Klassen, bereitzustellen, die schließlich als Objekte instanziiert worden sind – 15 Jahre später bei der Entwicklung von C++ als Idee aufgenommen. C++ ist eine Sprache, die teilweise entwickelt wurde, um UNIX-basierte Prozesse in Rechnernetzwerken zu handhaben,[12] was später maßgeblich zur Entwicklung einer weiteren OOP-Sprache, nämlich Java, beitrug. Ihre implizite Ontologie der Welt als Sequenz von Ereignissen (oder Aktionen), ihre eingebauten Gestaltungsrichtlinien sowie ihre Verbindungen mit unterschiedlichen gedachten

ein Feld, das aus der Kriegslogistik entstand. Dabei wurde z. B. der gesamte Managementprozess einer Versorgungskette so gedacht, als ob er alle Ebenen der Produktion und der Verwendung beinhalten würde. Das Ziel war es, bestimmte Arten von Effizienz zu maximieren. [Anm. der Übers.: Es wird hier die in den 1960er Jahren übliche Übersetzung «Unternehmensforschung» benutzt, um den arbeitsweltlichen Kontext deutlicher zu machen.]

8 Partizipatorisches Design ist selbst einem ständigen Wandel unterworfen, der zahlreiche widersprüchliche Tendenzen enthält, so dass es in einigen Fällen nur eine verfügbare Technik von vielen für das effektive Wissensmanagement und die Entscheidungsfindung in einem Projekt ist und das systematische Sammeln von relevanten menschlichen Faktoren ermöglicht, ohne eine kritische Analyse der Arbeit zu versuchen. Gleichzeitig bleibt es als eine hartnäckige Forderung und Vorbild für logisch aufgebaute Beziehungen zwischen den Objekten und Arbeitsprozessen und denjenigen, die mit ihnen arbeiten. Siehe z. B. Keld Bødker, Finn Kensing und Jesper Simonsen, *Participatory IT Design, designing for business and workplace realities*, Cambridge (MIT Press) 2004.

9 Vgl. Nygaard, Dahl, Development of the SIMULA Languages.

10 Mit anderen Worten werden die Werte aller Variablen gespeichert. Man stelle sich vor, ein Objekt namens «Bankkonto» würde wichtige Dinge wie «Saldo» oder «Kontoinhaber» usw. vergessen.

11 Anm. der Übers.: Diese Aussage ist fragwürdig. Zwar wurde SIMULA nie zu einer weit verbreiteten bzw. einer Standardsprache. Jedoch erfreute sich SIMULA insbesondere in den 1970er Jahren deutlicher Beliebtheit für kleine und große Projekte und wurde auf alle zu der Zeit existierenden Großrechnerarchitekturen portiert. 1973 wurde zudem die *Association of SIMULA Users* gegründet, welche bis heute besteht. Vgl. u. a. Holmevik, Compiling SIMULA, http://www.idi.ntnu.no/grupper/su/publ/simula/holmevik-simula-ieeeannals94.pdf, gesehen am 15.2.2012.

12 Siehe Bjarne Stroustrop, *A History of C++: 1979–1991*, Murray Hill, New Jersey (AT&T Bell Laboratories), o.J.

und konkreten Umsetzungen der Organisation von Arbeit sind Indikatoren für mögliche zukünftige Programmiertechniken.

Die Entwicklung von Smalltalk[13] am Xerox Palo Alto Research Center (PARC) in den 1970er und 1980er Jahren unter der Leitung von Alan Kay könnte als eine alternative Geschichte der Objektorientierung erzählt werden.[14] Smalltalk beschreibt das Wesen von Objekten hauptsächlich durch den Nachrichtenaustausch zwischen Entitäten oder Objekten innerhalb eines Systems. In der Tat sind es die Beziehungen *zwischen* Dingen, die Kay letztendlich als grundlegend für die Ontologie von Smalltalk ansieht.[15] Objekte werden als Instanzen von Idealtypen oder Klassen erzeugt, aber ihr tatsächliches Verhalten entsteht durch Nachrichten, die von anderen Objekten kommen, und es sind diese Nachrichten (oder Ereignisse – die Unterscheidung ist aus informatischer Sicht unwichtig), denen tatsächlich am meisten Bedeutung zukommt. Obwohl die effektive Anordnung, in der rechnerische Ereignisse ausgeführt werden, wesentlich linear ist, ist eine solche Reihenfolge nicht streng festgeschrieben, und es gibt einen umfassenden Sinn für eine Polyphonie von Ereignissen und Entitäten in dynamischer Beziehung. Kay liefert eine Begründung für die dynamischen Beziehungen zwischen den rechnerischen Objekten, die Smalltalk zu konstruieren versucht, indem er auf die extreme Starre und Unflexibilität von Nutzerschnittstellen damaliger Großrechner hinweist: Ein Ansatz, der von rechnerischen Objekten ausgeht, erlaubt dabei eine flexiblere Beziehung zwischen Nutzer und Rechner, eine Beziehung, die später von einem Kollegen Kays als Möglichkeit, das kreative Talent eines Individuums anzuzapfen, schöngefärbt wurde.[16] Diese flexible Beziehung zu sich dynamisch verbindenden Objekten ist Teil einer utopischen Vision der EDV, die sich schließlich im PC materialisiert; gemäß einer beliebten Darstellung wurde die Kontrolle den Institutionen, die Computer (typischerweise Großrechner) besitzen, abgerungen, um sie einer neuen Form ‹persönlicher Beherrschung› zu übergeben.

Von entscheidender Bedeutung für Smalltalk ist die in den Vordergrund tretende interaktive Qualität der Berechnung – ein großer Fortschritt gegenüber dem strikt vorgeschriebenen Ablauf von Befehlsfolgen vorhandener Sprachen. Genauer gesagt wird das rechnerische Objekt in Smalltalk nach seiner Leistungsfähigkeit betrachtet, sich in ein Lernsystem einzufügen – und zwar eines, das (Piaget, Vygotsky und anderen folgend) im Kern konstruktivistisch ist.[17] Als solches besteht das operative Ziel eines Objektes nicht nur darin, gut innerhalb der Domäne des Programms zu funktionieren, sondern auch darin, sich in die Lernprozesse des Nutzers einzuklinken, diese anzuregen und zu unterstützen – jedoch nicht (wenigstens nicht bewusst) zu prägen. Ebenso wie bei der Betonung des Nachrichtenaustauschs zwischen vielen einfach, aber präzise definierten Objekten als Basisstruktur des Programms, hebt Smalltalk solche Interaktion, die Beziehungen zwischen Objekten und zwischen Objektnachrichten und Nutzern hervor.

Das historische Argument, das hier gemacht werden kann, besagt, dass die Verbindung zwischen Smalltalk und interaktionsbasiertem Lernen im Vergleich

13 Anm. der Übers.: Warum Kays Beitrag zur Entwicklung der objektorientierten Programmierung als Alternativentwicklung dargestellt wird, ist aus dem Text nicht offensichtlich. Kay, der den Begriff «object-oriented programming» geprägt hat, bezog SIMULA durchaus in die Entwicklung seiner Sprache Smalltalk mit ein.

14 Kay ist ein Forscher, dessen wissenschaftliche Prägung auf Ivan Sutherland und die Erfindung von *Sketchpad* – das erste CAD-System, dem ein «Konzept» von bestimmten digitalen Objekten innewohnte – zurückreicht. Um den Einfluss seiner Arbeitsgruppen zu verstehen, muss man nur die zeitgenössische Computerkultur überblicken. Siehe u. a. den Überblick zu Smalltalk in der Sonderausgabe der Zeitschrift *Byte*, Vol. 6, No. 8, August 1981.

15 Alan Kay, Prototypes Versus Classes, in: *Squeak Developers Mailing List*, 10.10.1998, http://lists.squeakfoundation.org/pipermail/squeak-dev/1998-October/017019.html, gesehen am 15.2.2012.

16 Siehe Daniel Inglis, Design Principles behind Smalltalk, in: *Byte*, Vol. 6, Nr. 8, August 1981.

17 Eine interessante Vergleichsmöglichkeit findet man in den Arbeiten von Seymour Papert, dem Entwickler der Programmiersprache LOGO – einer Sprache, die speziell zu Unterrichtszwecken entworfen wurde. Bei beiden Sprachen standen die epistemischen Folgen für die Nutzer im Vordergrund. Beide Sprachen werden, in ihren verschiedenen Inkarnationen, durch eine aktive Nutzergemeinschaft gepflegt.

zu den Versuchen, Wissen über Arbeitsstätten mit frühen Programmversionen von SIMULA zu simulieren, einen Wandel markiert. Aber es ist ein Wandel, der von Kontinuitäten und Diskontinuitäten geprägt ist, denn obwohl die Ziele der Programmierung unterschiedlich sind, ist die Ambivalenz, die in SIMULA durchscheint, in Smalltalk immer noch vorhanden. Diese Ambivalenz besteht zwischen einem Programm als einem epistemisch adäquaten Modell eines Prozesses bzw. eines Sets von Prozessen und einem Programm, das selbst ein materiell sehr effektives Set von Prozessen ist; eine Ambivalenz, die man oberflächlich zusammengefasst als *analytische* und *synthetische* Funktionen des Rechnens auffassen kann und die auch in Smalltalk bestehen blieb. Deshalb nun eine kurze Erklärung. Über die Entwicklung von Smalltalk bemerkte Kay: «object-oriented design is a successful attempt to qualitatively improve the efficiency of modelling the ever more complex dynamic systems and user relationships made possible by the silicon explosion.»[18]

Die Modellierung kann man hier nicht wirklich als eine einfache Repräsentation verstehen, denn wenn diese komplexen Systeme und Nutzerbeziehungen, die objektorientiertes Design modelliert, überhaupt erst durch die Verbreitung von Mikroprozessoren *(silicon explosion)* möglich werden, dann begeben wir uns in einen Bereich ontologischer Kunstfertigkeit: Es handelt sich nicht einfach um das Ansammeln von Wissen über die Welt durch Artefakte, sondern vielmehr um das Erschaffen von Modellen von Dingen, die es sonst überhaupt nicht gibt. Die Unterscheidung ist wichtig, wird aber oft übersehen, weil sie auf die Grenzen verweist, die der Untersuchung rechnerischer Objekte innerhalb der epistemologischen Konzepte von Repräsentation gezogen sind. Ein Verständnis des Rechnens auf der Grundlage der zweiten Sichtweise ist weit verbreitet, und tatsächlich legt Kay selbst nahe: «everything we can describe can be represented by the recursive composition of a single kind of behavioural building block that hides its combination of state and process inside itself and can be dealt with only through the exchange of messages.»[19] So gesehen entwickelt ein in einer objektorientierten Sprache geschriebenes Computerprogramm eine Art intentionale Logik in dem Sinn, dass die Objekte, aus denen das Programm besteht, eine interne konzeptionelle Struktur aufweisen, die seine Beziehung zu dem bestimmen, was es ‹bezeichnet› *(refers to)*, und damit suggeriert, dass Objekte tatsächlich Konzepte seien – Konzepte, die Objekte innerhalb einer maschinischen Materialisierung eines logischen Kalküls repräsentierten.[20] Als in der physikalischen Raumzeit materialisierte Konzepte effektiver Rechenprozesse, die gleichwohl Objekte außerhalb der Maschine bezeichnen, zeigen Smalltalk-Objekte deutlich die ‹analytische› Funktion des Rechnens. Doch insofern Smalltalk-Software selbst eine Interaktionsmöglichkeit darstellt, ist ihre ‹synthetische› Funktion gleichermaßen offensichtlich.

Auf der einen Seite haben wir also die Weise, in der objektorientierte Programmierung behauptet, die Dekomposition von Prozessen in rechnerisch-

18 Ebd.

19 Alan Kay, The Early History of Smalltalk, auf: http://www.smalltalk.org/smalltalk/TheEarlyHistoryOfSmalltalk_Abstract.html/, gesehen am 15.2.2012.

20 Kay bemüht zu diesem Punkt Carnap, dessen Aufsatz «Meaning and Necessity» von 1947 eine anspruchsvolle Entfaltung der Begriffe Intension und Extension in der Logik bietet.

adäquate ‹Repräsentationen› von Entitäten (= Objekten) zu ermöglichen. Auf der anderen Seite steht die schlichte Tatsache, dass ein rechnerisches Objekt selbst eine eigenständige Entität ist, eine Entität mit einer spezifischen, materiellen Form, die mit anderen Entitäten verknüpft ist (ob rechnerisch oder nicht), die eine irgendwie geartete materielle ‹Repräsentation› in der Software aufweisen mögen, oder auch nicht. Es ist – wie wir behaupten – dieser zweite Aspekt rechnerischer Objekte, das Verhalten *mit* anstatt *von* Objekten zu modellieren, den es genauer zu verstehen gilt.[21]

Abstraktion, Fehler und die Erfassung von Handlungsmacht

Es wird oft behauptet, dass durch die Entwicklung der objektorientierten Programmierung eine neue Ära der *Interaktion* zwischen Menschen und Maschinen ermöglicht wurde. Darin steckt ein Körnchen Wahrheit – nicht nur aufgrund der Art, wie die Architektur der Beziehungen zwischen den einzelnen Objekten eine Programmstruktur überflüssig macht, die eine strikt vorgeschriebene Abfolge von Aktionen benötigt. Wie dem auch sei, das Konzept der Interaktion als Beschreibung einer *reziproken* Relation zwischen zwei unabhängigen Entitäten ist für den Versuch, die historische Genese und die eigentümliche Verstrickung von Beziehungen zwischen Computern und Menschen zu verstehen, manchmal unzureichend. Es wäre angemessener, die Beziehungen im Sinne einer Reihe von Formen der abstrahierungsfähigen Erstellung, *Erfassung* und Kodifizierung von Handlungsmacht zu betrachten: Ein Mausklick, ein Tastendruck, Dateneingabe, *affective investments* usw. Durch den Verweis auf die Entstehung und Erfassung von Handlungsmacht versuchen wir, zwei Dinge hervorzuheben: erstens, dass rechnerische Objekte nicht einfach vorgefertigte Kapazitäten oder Fähigkeiten anzapfen – vielmehr generieren sie neue Arten von Handlungsmacht, die dem Vorangegangenen ähnlich sein können, aber trotzdem unterschiedlich sind (eine Schreibmaschine, eine Tastatur und ein Keypad erfassen z. B. die Handlungsmacht der Finger in einer subtil unterschiedlichen Art und Weise); zweitens, dass die dadurch entstandene Handlungsmacht Teil einer asymmetrischen Beziehung zwischen Mensch und Computer ist, einer Art Kultivierung oder Einimpfung eines *maschinischen Habitus*, einem Set von Bestimmungen, die untrennbar mit den Technologien verbunden ist, die diese Ambivalenz kodifizieren und ihr Ausdruck verleihen. Eine detaillierte Untersuchung, weshalb und wieso diese Asymmetrie besteht, würde den Rahmen dieses Aufsatzes sprengen; für die Entwicklung eines angemessen konkreten Verständnisses der soziotechnischen Qualität der gegenwärtigen Machtverhältnisse wäre sie jedoch von entscheidender Bedeutung.

Ein Teil der Rhetorik um das interaktive Rechnen insistiert auf dem ‹intelligenten›, ‹reagierenden› Wesen von Rechengeräten, doch dies verschleiert die Dynamik der Softwareentwicklung und die partielle, additive Qualität der Entwicklung interaktiver Möglichkeiten. Außerdem sind Computer nicht beson-

21 Eine philosophische Referenz mag dies verdeutlichen. Das Konzept der Synthese, auf das wir verweisen, sollte nicht im Kant'schen Sinn (wie in *Kritik der reinen Vernunft* beschrieben) der Synthesen verstanden werden, bei denen es sich immer noch um die Repräsentation mit dem Menschen im Mittelpunkt dreht. Die passende Referenz ist stattdessen Whitehead, dessen Prehension ein unabhängig von darstellerischen Voraussetzungen funktionierendes Prinzip der Synthese bieten und eine ‹Emergenz› von Subjekten als Teil von ‹verwachsenen› Prozessen erlaubt, von denen jedwede Art von Prehension ein Teil ist. Vergleiche im dritten Teil von Alfred North Whitehead *Process and Reality* zur Theorie der Prehension. Eigentlich schließt das rechnerische Erfassen der Wirkung Prehension *und* Ingression mit ein, in der Art, dass ein rechnerisches Objekt dem Erfassten eine (kodierte) Form von Bestimmtheit verleiht. Ingression nach Whitehead besitzt Ähnlichkeiten zum *‹recording›* in den frühen Werken von Deleuze und Guattari. Siehe auch Steven Shaviro, *Without Criteria: Kant, Whitehead, Deleuze and Aesthetics*, Cambridge (MIT Press) 2009.

ders gut darin, Interaktionen zu *reparieren*. Im Alltag neigen sie dazu, erheblich unnachgiebiger als ihre Nutzer zu sein und weigern sich stumpfsinnig, eine Datei zu speichern, eine Webseite zu laden oder sogar komplett herunterzufahren. Auch wenn Systemabstürze heutzutage nicht mehr so häufig vorkommen wie früher, zeigt die Alltagserfahrung bei der Entwicklung von Mensch-Rechner-Interaktionen, dass Menschen gezwungen werden, einen erheblichen Teil ihrer Zeit damit zu verbringen, wie ein Computer denken zu lernen, Notlösungen zu entwickeln, sich den Vorgaben des Rechners anzupassen. In diesem Sinne haben ‹Bugs› eine wichtige Rolle dabei gespielt, asymmetrische Beziehungen zwischen Menschen und Maschinen aufzubauen. Menschen passen sich den Beschränktheiten des Computers verhältnismäßig schneller an – oder lernen zumindest, diese nicht mehr wahrzunehmen – als der Computer an den Menschen. Das liegt teilweise daran, dass die Zeit bis zum Erscheinen einer neuen Softwareversion (inkl. Fehlerbehebungen) viel länger ist als die zwischen individuellen Interaktionen mit der Software. Diese Asymmetrie suggeriert einen gewissen strategischen Wert der Beschränktheit von Maschinen, einer Beschränktheit, die Maschinen eine entscheidende Rolle bei der *Modellierung* des Nutzers zuschreibt, mit dem sie interagieren.[22]

In jedem Fall und bei allem Respekt gegenüber Kay: Ein rechnerisches Objekt ist ein *partielles* Objekt – seine definierenden Eigenschaften und Methoden sind notwendigerweise eine selektive und kreative Abstraktion spezieller Eigenschaften der zu modellierenden Sache, die zudem seine Interaktionsmöglichkeiten festlegt. Die Objekte – Textfelder, Listen, Hyperlinks usw. –, die eine Webseite ausmachen, bestimmen z. B. mehr oder weniger genau, was ein Nutzer damit tun kann. Dies ist nicht nur eine Funktion der Gestaltung und des Aufbaus der Seite, sondern, entscheidender, einer Bandbreite von vorher definierten Mengen kodierter Funktionalität. Jedes dieser Objekte und seine Entwicklung haben eine Geschichte,[23] was bedeutet, dass die Parameter der Interaktion durch eine Folge mehr oder minder erfolgreicher *Abstraktionen* von eigentümlich zusammengesetzter, mehrschichtiger und stratifizierter Art bestimmt sind.

Es ist wichtig hier festzuhalten, dass Abstraktion ein kontingenter, *realer* Prozess ist. Die für selbstverständlich erachteten Weisen, in denen Menschen mit Maschinen interagieren, sind das Produkt materieller Arrangements, die nicht immer bestehen.[24] Zugespitzt gesagt, werden diese realen Abstraktionen konkret und unablässig durch die Interaktion zwischen und mit dem Rechnerischen re-aktualisiert. Solche Abstraktionsprozesse könnte man eher als Formen der *Deterritorialisierung* – im Sinne von Deleuze und Guattari – verstehen, als einen Prozess, an dem «at least two terms» beteiligt sind, bei denen «each of the two terms reterritorialises on the other. Reterritorialisation must not be confused with a return to a primitive or older territoriality: it necessarily implies a set of artifices by which one element, itself deterritorialised, serves as a new territoriality for another».[25]

22 Diese Art von Modellierung kann hier, wegen der Auswirkungen auf den Existenzmodus der Nutzer – ihre Gewohnheiten und Erwartungen prägend –, als ontologisch verstanden werden. Dies ist, im Gedanken an eine epistemologische Problematik der Repräsentation, etwas schwer zu ‹sehen›. Wir behandeln diese Frage im Fazit weiter.

23 In Anlehnung an die Biologie könnte man sagen, dass die Geschichte sowohl onto- als auch phylogenetisch ist.

24 Das Konzept der realen Abstraktion wird mit Marx und dem Marxismus assoziiert. Siehe dazu insbesondere die Arbeiten von Alfred Sohn-Rethels zur geistigen Arbeit und Handarbeit und dessen Erörterung bei Alberto Toscano. Whitehead, Deleuze und Guattari ermöglichen es uns, ein präzises Verständnis dieses Prozesses zu entwickeln.

25 Gilles Deleuze, Félix Guattari, *A Thousand Plateaus*, Minneapolis (Univ. of Minnesota Press) 1987, 193.

In diesen Begriffen gedacht, verbindet die Erfassung von Handlungsmacht die formale Strukturierung von rechnerischen Objekten mit den umfassenderen Prozessen, deren Teil sie sind, was es uns im Gegenzug erlaubt: a) genauer zu spezifizieren, dass die formale Strukturierung und Komposition von Objekten eine vektorielle Qualität besitzt, und b) die entsprechenden Eigenschaften der Re-Territorialisierung genauer zu behandeln. Im Sinne eines realen Prozesses ist ein Abstrahieren *von* gleichermaßen ein Abstrahieren *zu*: Eine Abstraktion ist nur unter der Bedingung wirksam, dass sie ein Teil einer größeren Menge von Relationen ist, über die und durch die sie stabilisiert und befestigt wird. Diesen Sachverhalten wenden wir uns nun genauer zu.

Das Milieu stabilisieren

Theoretisch kann alles, was man in einer Programmiersprache berechnen kann, genauso auch in einer anderen berechnen: Das ist eine der Lektionen des Turing'schen Konzepts der Universalmaschine. Etwas nüchterner gesagt und auf das vorliegende Problem bezogen, bedeutet dies, dass eine objektorientierte Programmiersprache den Ingenieuren, die mit ihr arbeiten, ein Set von Designeinschränkungen *(constraints)* bereitstellt, indem sie bestimmte programmatische Konstrukte bevorzugt sowie bestimmte Lösungsansätze technischer Probleme über andere stellt. Bei dem Versuch, die Dynamik zu beobachten, die die materielle Textur der Softwarekultur regiert, ist die Existenz solcher Designeinschränkungen besonders wichtig. Die Frage lautet dann: Gibt es angesichts der Art, wie Asymmetrien in Mensch-Maschine-Beziehungen die Erfassung von Handlungsmacht erlauben, eine Möglichkeit, wie die Verbreitung und Erweiterung dieser Relationen berücksichtigt werden kann? Lässt sich in den Arbeitspraktiken mit rechnerischen Objekten etwas entdecken, das beim Verständnis dieser Dynamik helfen könnte?

Eine Eigenschaft, welche insbesondere der objektorientierten Programmierung nachgesagt wird, ist, dass sie die *Wiederverwendung* von Code erleichtere. Anstatt immer wieder den gleichen oder ähnlichen Code für unterschiedliche Programme zu schreiben, spart es Zeit und Arbeit, den Code einmal zu schreiben und dann in verschiedenen Programmen wiederzuverwenden. Wiederverwendbarkeit beschränkt sich nicht nur auf die objektorientierte Programmierung; sie ist in den Funktionalitäten jeder Art von Programmiersprachen in dem banalen Sinn angelegt, dass, wenn man beispielsweise eine bestimmte Variable innerhalb einer Routine initiiert (sagen wir eine Gleitkommazahl), durch das Kompilieren einem Wert eine Adresse im Speicher zugewiesen wird. Dies ist keine Aufgabe, die die Programmiererin bzw. der Programmierer selbst erledigen müsste – die Routinisierung und Automatisierung rechnerischer Aufgaben ist ein grundlegendes Merkmal des Betriebs. Allerdings begünstigt die objektorientierte Programmierung die Wiederverwendbarkeit von Code für rechnerisch abstrakte Arten von Entitäten und Operationen – mit anderen

Worten für Entitäten und Operationen, die sich unmittelbarer auf die Herstellung der Verbindung zwischen dem Computer und der Außenwelt beziehen. *Klassenbibliotheken* sind dafür bezeichnend. Obwohl sie nicht ausschließlich bei objektorientierten Sprachen vorkommen, stellen sie eine vorgefertigte Menge von Objekten mit vordefinierten Methoden, Eigenschaften usw. bereit, die in Programmiersituationen umfassend Verwendung finden: In der Programmiersprache Java beispielsweise beinhaltet die Java-Ein-/Ausgabebibliothek ein ‹File›-Objekt, das der Programmierer benutzen kann, um Standardoperationen auf einer programmexternen Datei (Daten lesen, Daten schreiben usw.) auszuführen.[26] Die Wiederverwendung von Code allgemein legt nahe, dass die Umstände in denen der Code situiert ist, die Zwecke zu denen er Anwendung findet, die Interaktionen die er hervorruft, und das Verhalten, welches er auslöst, einigermaßen geregelt und stabil sind. Dies legt, anders gesagt, nahe, dass die typischen Arten von Software an der Entstehung ihrer eigenen ökologischen *Nischen* beteiligt waren.

Die Möglichkeit der Wiederverwendbarkeit muss daher aus zwei Blickwinkeln zugleich betrachtet werden: 1) als etwas, das eine spezielle Aufforderung *(affordance)* innerhalb der Struktur einer objektorientierten Sprache besitzt; und 2) als etwas, das in seinem Kontext die Chance bekommt, Wurzeln zu schlagen, sich zu stabilisieren, ein Territorium zu besetzen. Im vorherigen Abschnitt wurde nahe gelegt, dass die einfache Dynamik der Adaption bzw. Habituation für Letzteres verantwortlich ist. Ersteres kann in den technischen Funktionalitäten der objektorientierten Programmiersprachen verortet werden. Wir werden uns zunächst dieses Themas annehmen, bevor wir uns einer umfassenderen Betrachtung stabilisierender Praktiken zuwenden.

Eines der wichtigsten Alleinstellungsmerkmale der objektorientierten Programmierung ist der Einsatz von *Vererbung*. Ein rechnerisches Objekt im Sinne der Objektorientierung ist eine Instanziierung einer *Klasse*, ein programmatisch definiertes Konstrukt, ausgestattet mit bestimmten Eigenschaften und Methoden, die es ihm erlauben, bestimmte Aufgaben zu erledigen. Diese Eigenschaften und Methoden sind kreative Abstraktion. Vererbung ist eine Eigenschaft, die oft – wenngleich fälschlicherweise – semantisch als eine ‹ist ein›-Beziehung charakterisiert wird: Eine Perserkatze bzw. eine Siamkatze *ist eine* Katze, ein Sparkonto *ist ein* Konto usw. Das Vererbungsprinzip konstituiert eine Hierarchie von Objekten, die oftmals durch die Begriffe Klassen und Subklassen beschrieben wird. Wie diese Hierarchie zu verstehen ist, ist wiederum selbst eine komplexe Frage (trotz Versuchen der Informatiker, das Vererbungsprinzip zu formalisieren). Entscheidend ist, dass das Vererbungsprinzip Programmierern ermöglicht, auf existierende rechnerische Objekte mit relativ gut bekanntem Verhalten aufzubauen, indem dieses Verhalten durch neue Methoden und Eigenschaften erweitert wird.

Das Vererbungsprinzip unterstellt eine Situation, in der Objekte ihren Geltungsbereich, ihr Territorium, durch kleine Variationen, durch inkrementelle

26 Anm. d. Übers.: Dafür werden zusätzliche Objekte, sogenannte *Input-* bzw. *OutputStreams*, benötigt.

Ergänzungen, so erweitern, dass die Erwartungen, wie Objekte sich zu verhalten haben, eher bestätigt und nicht gestört werden. Grob gesagt ist es einfacher zu erben und zu erweitern und *anzunehmen*, dass kleine Unterschiede nur Abweichungen von der Norm sind, als zu erwägen, dass solche Abweichungen eventuell Hinweise auf eine andere Ausgangssituation – eine andere Welt – sind. Verhaltensmodellierung anhand der technischen Beschränkung der Vererbung stabilisiert die in der Software erfassten Beziehungen und Praktiken.

Entwurfsmuster *(design patterns)* erweitern diese Logik der Wiederverwendung von Code auf eine komplexere Menge von Algorithmen, die einen breiteren Problemkreis abdecken. Das Konzept der Entwurfsmuster stammt von dem Architekten Christopher Alexander, für den solche Muster die Beschreibung von Problemen sind, die «occur over and over again in our environment.»[27] Ein Software-Entwurfsmuster bietet eine wiederverwendbare Lösung zu einem informatischen Problem (wir geben dafür gleich ein Beispiel), und obwohl ein solches Muster selbstverständlich eine technische Entität ist, ist es zugleich auch eine partielle Übertragung eines ursprünglich nicht-rechnerischen Problems. Ein Geschäfts-Informationssystem beispielsweise, das entworfen wird, um einen Lagerbestand zu kontrollieren und das auf einem ‹just-in-time›-Modell der Bestandskontrolle basiert, wird Gestaltungsprobleme bezüglich einer Menge von Relationen zwischen rechnerischen Objekten nach sich ziehen, die anders sind, als bei einem System, das auf eine eher herkömmliche Bestandskontrolle ausgerichtet ist (bei der eine größere Menge eines Gegenstandes zu niedrigeren Stückkosten erworben wird), weil ein solches System ganz andere Aufgaben zu erledigen hätte. Es impliziert ein veränderliches Set von Beziehungen zwischen Software und Nutzern, die möglicherweise eine automatisierte Menge von Links zwischen einer Firma und Firmen weiter oben in der Lieferkette zur Folge hat.

In einer Hinsicht sind Entwurfsmuster eine Antwort auf das Kernproblem der objektorientierten Softwareentwicklung: die Analyse, Dekomposition und Modellierung der Programmaufgabe als ein Set von Objekten mit wohldefinierten Eigenschaften. In der Tat werden Entwurfsmuster üblicherweise so von Softwareentwicklern verstanden. Aber schon ihre bloße Existenz ist interessant, und zwar nicht nur, weil sie einen Beweis für die steigende Komplexität der Rechnerumgebung liefern, sondern auch wegen ihrer Stabilität und Regularität – Eigenschaften, die diese Muster im Gegenzug produzieren. Solche materiellen Voraussetzungen werden bei Diskussionen zur objektorientierten Programmierung (oder sogar zur Programmierung im Allgemeinen) üblicherweise nicht berücksichtigt, in denen der selbstredende Vorteil und die vernünftige Selbstverständlichkeit eines Denkens in objektorientierten Begriffen in Lehrbüchern üblicherweise durch die Analogie zur Selbstverständlichkeit von Objekten selbst gefördert wird wie beispielsweise durch die Aufzählung all der eher stabilen Objekte, die sich im Büro des Autors befinden – Tische, Stühle, Papiere, Bücher und die Regale, in denen sie stehen.[28] Und doch ist die Stabilität

27 Christopher Alexander, zitiert in Erich Gamma u. a., *Design Patterns. Elements of Reusable Object-Oriented Software*, Indianapolis (Addison-Wesley) 1995, 2.

28 Stephan Goldsack, Stuart Kent, *Formal Methods and Object Technology*, Wien (Springer) 1996.

einer Umgebung absolut entscheidend für den effektiven Einsatz rechnerischer Objekte.[29] Wie dem auch sei, wird (wie wir eingewendet haben) die pädagogische Betonung des relativ Einfachen – und die daraus folgende Aufmerksamkeit für den Vorrang der epistemologischen Dimensionen rechnerischer Objekte – nicht den Prozessen gerecht, die in der historischen Entwicklung der Softwarekultur am Werk sind. Insbesondere trägt sie wenig zum Verständnis der Art und Weise bei, wie Handlungsmacht nicht nur erfasst, sondern auch in vorhersehbaren, routinemäßigen Mustern konfiguriert und stabilisiert wird. Es wäre vielleicht angemessener, die stabile, einfache und scheinbar selbstredend gegebene Qualität von rechnerischen Objekten als das Ergebnis einer komplexen soziotechnischen Genese zu betrachten.

Datenkapselung, Ausnahmen und Unerkennbarkeit

Bisher haben wir versucht, die materiellen Aspekte jener komplexen Prozesse der Abstraktion zu thematisieren, die in der objektorientierten Programmierung am Werk sind. Trotzdem müssen wir darauf bestehen, dass rechnerischen Objekten eine kognitive Funktion zukommt. Allerdings wird diese Funktion hauptsächlich durch die oft blinde und im Dunkeln tappende Art der Gestaltung nicht-rechnerischer Prozesse erfüllt.

Rechnerische Objekte operieren in einer Welt, in welcher das Zusammenspiel zwischen ihrer privaten internen Funktionsweise und ihrer öffentlichen Fassade durch präzise definierte, vertragsförmige Schnittstellen festgelegt wird. Man interagiert mit der Maschine nicht irgendwie, sondern mit einem gewissen Grad an Freiheit und Spielraum, der sehr präzise, programmatisch spezifiziert ist. *Datenkapselung* – oft als eines der Grundprinzipien der objektorientierten Programmierung bezeichnet – erzwingt, zusammen mit dem Konzept der *Ausnahme (exception)*,[30] eine strikte Trennung zwischen Innen und Außen, die nur durch das sorgfältige Entwerfen der Schnittstellen überbrückt werden kann, wodurch ‹Nichtwissen› zu einem zentralen Prinzip der Gestaltung wird. Der Begriff Datenkapselung bezeichnet dabei die Art, wie objektorientierte Programmiersprachen sowohl die Daten, die den Zustand *(state)* der Objekte beschreiben, die ein Programms ausmachen, verstecken als auch die Details der Operation, die das Objekt ausführt.[31] Um auf die Daten, die den Zustand eines Objekts beschreiben, zuzugreifen oder diese zu modifizieren, verwendet man typischerweise eine *get*- oder *set*-Methode. Diese Methoden machen die Interaktion, die durchgeführt wird, explizit sichtbar. Datenkapselung bietet – auf der Ebene des formalen Gebäudes der Programmiersprache – eine Variante eines von Programmierern beachteten generellen Prinzips. Dieses besagt, dass man bei der Programmierung einer Schnittstelle zu einem Programmelement immer die ‹Implementierungsdetails› verstecken soll, sodass die Nutzer nichts über die Daten wissen, die für das Funktionieren entscheidend sind, und auch nicht in Versuchung geraten, diese zu modifizieren. Den Begriff ‹Nutzer› muss man in

29 Dieses Thema wird in Isabelle Stengers *Thinking with Whitehead: A Free and Wild Creation of Concepts* (Cambridge [Harvard Univ. Press] 2011) untersucht: die ‹Geduld› der Umgebung ermöglicht dabei den infektiösen Dynamiken der Macht wirksam zu operieren.

30 Anm. der Übers.: Das Konzept der *exceptions* ist nicht nur ein Merkmal der objektorientierten Sprachen und ist eher als Methodologie und Entwicklungsparadigma zu verstehen, welche auch in anderen Programmiersprachen Anwendung finden.

31 Nicht alle Informatiker bzw. Softwareentwickler sind der Meinung, dass Datenkapselung und *information-hiding* bzw. *data-hiding* dasselbe sind. Wir brauchen uns um die Details der Auseinandersetzung an dieser Stelle nicht zu kümmern.

diesem Fall in Bezug auf die betrachteten Programmobjekte verstehen – Programmierer erlauben Nutzern von Webseiten nicht, den Code auf einer Webseite, der in einem Browser ausgeführt wird, nach Belieben anzupassen, oder eine Gruppe von Programmierern wird vielleicht die Implementierungsdetails einer Menge rechnerischer Objekte vor anderen Programmierern verstecken, die diese benutzen wollen usw.

Es heißt, dass die Datenkapselung neben der Förderung der Wiederverwendung von Code das Risiko von fehlerhaftem Code minimiere, welcher durch unfähige Programmierer verursacht wird, die Zugriff auf Daten haben und diese auf eine Weise verändern, die dazu führen könnte, dass Objekte sich auf unerwartete Weise verhalten. Eine zentrale Maxime für Programmierer ist, dass man immer ‹defensiv› programmieren soll, immer ‹sicheren› Code schreiben soll, und sogar akzeptieren muss, dass Eingaben – egal auf welcher Ebene man dies definieren mag – immer ‹böse› sind.[32] Ein stark reguliertes Zusammenspiel zwischen dem Innenleben und der äußeren Funktionsweise von Objekten ermöglicht es, den stabilen Betrieb von Software zu gewährleisten. Dies ist wohl Teil einer historischen Tendenz und eines proprietären Trends, die Nutzer vom Innenleben der Maschinen fernzuhalten, der zu einem komplexen sozio-technischen Wirrwarr von geistigem Eigentum, Risikomanagement und Arbeitsteilung führt,[33] dessen Ergebnis ist, dass der Zugang des Programmierers zu tieferen Operationsebenen eingeschränkt wird (obwohl das Schreiben von Code theoretisch leichter werden sollte).

Als Grundsatz und als technische Einschränkung bilden Datenkapselung und das Verstecken von Daten zumindest eine techno-ökonomische Hierarchie, innerhalb derer der die Produzenten von Programmiersprachen die Richtung von Innovation und Veränderung kontrollieren können, indem sie ein *lock-in* vorantreiben und die Arbeitsteilung so strukturieren, dass Programmierer bestärkt werden, proprietäre Klassenbibliotheken zu verwenden[34] anstatt ihre eigenen zu entwickeln. Die Entwicklung neuer Formen des Wissens durch Maschinen hat eine bestimmte, heute globale Arbeitsteilung zwar erst ermöglicht, wird zugleich aber von einer technisch beschränkten, normativen Anschauung darüber verhindert, was Programmieren sein soll.

Obwohl offensichtlich viele weitere Faktoren bei der Gestaltung von Programmierpraktiken eine Rolle spielen, ist die tiefsitzende Gewohnheit, Klassenbibliotheken zu verwenden, eindeutig etwas, das aus der technischen Aufforderung *(affordance)* der Datenkapselung resultiert. Eine viel feingliedrige Arbeitsteilung wird möglich, wenn die Softwareentwicklung des Systems bzw. der Anwendung in diskrete ‹Brocken› aufgeteilt werden kann. Jede Klasse oder Klassenbibliothek (aus welcher Objekte abgeleitet werden) kann dabei von einem anderen Programmierer oder einer Gruppe von Programmierern erstellt werden, wobei die Details der Klassenoperationen von anderen Projektteams gefahrlos ignoriert werden können. Der gegenwärtige Trend zur Globalisierung der Softwareentwicklung, mit seinen delokalisierenden Metriken der

32 Siehe Alfred Tarski, Truth and Proof, in: *Scientific American*, Juni 1969. Nachgedruckt in R.I.G. Hughes, *A Philosophical Companion to First-Order Logic*, Cambridge (Hackett) 1993. Für eine weiterführende Diskussion, siehe Matthew Fuller, Andrew Goffey, *Evil Media*, Cambridge (MIT Press) 2012.

33 Dies ist eine Lesart des ziemlich deterministischen Arguments von Friedrich Kittler in seinem Aufsatz: Protected Mode, in: Norbert Bolz, Friedrich Kittler, Georg-Christoph Tholen (Hg.): *Computer als Medium*, München (Fink) 1994, 209–220.

34 Anm. der Übers.: Die Darstellung von Datenkapselung als ‹vendor lock-in› ist etwas ungewöhnlich. Klassenbibliotheken finden auch große Anwendung in der Welt der Open-Source-Software-Entwicklung. Sie erlauben Entwicklern auf bereits erprobten Code aufzubauen. Das Prinzip bereits einmal gelöste Probleme in Bibliotheken / Pakete auszulagern und diese wiederzuverwenden – anstatt ein bekanntes Problem immer wieder neu zu lösen – ist ein zentrales Paradigma bei Softwareentwicklern. Ursprünge finden sich u. a. bereits in der Hackerkultur rund um den *Tech Model Railroad Club* (TMRC) des *Massachusetts Institute of Technology* (MIT) in den 1960er Jahren.

Produktivität, hätte ohne die Aufteilung der Arbeit in Brocken, welches durch Datenkapselung begünstigt wird, nicht das aktuelle Ausmaß erreicht.[35]

Zuletzt wollen wir einen kurzen Blick auf die *Ausnahmebehandlung (exception handling)* erfen. Während Datenkapselung dafür zuständig ist, stabilisierende Abstraktionen zu schaffen, indem sie das Zusammenspiel des Inneren und Äußeren eines rechnerischen Objektes regeln und bestimmen, was Objekte voneinander wissen können, beschreibt Ausnahmebehandlung die Art, wie rechnerische Objekte auf alles Unerwartete reagieren. Ein Programm und dessen Objekte operieren immer nur innerhalb eines bestimmten Sets von Parametern, das mögliche Verbindungen des Objekts mit seiner Umgebung definiert und Annahmen darüber enthält, worauf das Programm in dieser Umgebung stoßen könnte. Wenn diese Annahmen nicht erfüllt werden (beispielsweise wegen eines fehlenden Browser-Plugin oder weil ein wichtiger DLL [*Dynamic-Link-Library*, eine wichtige Programmbibliothek, Anm. d. Übers.] beim Entfernen einer unerwünschten Anwendung gelöscht wurde), wird das Programm nicht wie erwartet funktionieren. Die Ausnahmebehandlung bietet einen Weg, den Kontrollfluss eines Programms aufrechtzuerhalten, obwohl bestimmte Annahmen nicht erfüllt werden konnten, wodurch sichergestellt wird, dass eine Anwendung bzw. das System nicht abstürzen muss, bloß weil irgendein unvorhergesehenes Problem aufgetreten ist. In der objektorientierten Programmierung ist eine Ausnahme ein Objekt wie jedes andere – eines, aus dem man einen Subtyp erzeugen bzw. dessen Funktionalität erweitern kann usw.[36]

Technisch gesehen sind die Prinzipien, gemäß derer man Ausnahmen verwendet, einsichtig, und indem man sie als Objekte mit allem, was dazugehört, behandelt, wird ihre programmatische Handhabung vereinfacht. Weniger gut verstanden ist allerdings die Art, wie die Methoden der Ausnahmebehandlung selbst die Beziehungen der rechnerischen Objekte mit der Außenwelt materiell gestalten. Vom Standpunkt der rechnerischen Objekte betrachtet, ist die Welt im Allgemeinen eine riesige und größtenteils unbekannte Ansammlung von Ereignissen, *zu* der solche Objekte nur unter sehr eingeschränkten Bedingungen Zugang haben können, aber auch *in* welcher Objekte nur sehr begrenzt Belange haben. Es ist die Aufgabe des Programmierers, alles Relevante so präzise wie möglich zu spezifizieren. Dies kann auf vielfältige Arten erreicht werden – z. B. garantiert die Praxis des ‹Validierens› von Nutzereingaben (durch eine Strukturüberprüfung, um festzustellen, ob die Eingaben einem bestimmten ‹regulären Ausdruck› gerecht werden), dass die für die Eingabe zuständigen rechnerischen Objekte keine unerwarteten Eingaben erhalten, die sie nicht verarbeiten können (wie beispielsweise ein falsches Datumsformat).

Weil die Verwendung von Ausnahmebehandlungen in einem Programm rechnerischen Objekten ermöglicht, ohne größere Störungen ihrer Arbeit nachzugehen, und weil ihr Status als rechnerische Objekte es erlaubt, sie wie alle anderen Objekte zu behandeln,[37] wird das Erfordernis, der Problemursache der Ausnahme nachzugehen (Systemanalyse und Entwurfsentscheidungen, die

35 Die globale Arbeitsaufteilung wird diskutiert in Audris Mockus, David M. Weiss, Globalization by Chunking, a quantitative approach, in: *IEEE Software*, März/April 2001, 30–37. Ungeachtet der durch die technosoziale Genese der Objekte implizierten Spannungen, ist dies in diesem Zusammenhang selbst ein forschungswürdiges Thema. Es ist besonders lehrreich, die Verteilung und Verwendung von Codemodulen in der Scriptsprache *Perl* und ihrer dazugehörigen proprietären Klassenbibliotheken mit einer Sprache wie Microsofts C# zu vergleichen. Es ist ebenfalls erwähnenswert, dass einer der Kritikpunkte an der objektorientierten Programmierung ist, dass sie keinerlei Fachkenntnisse erfordert. Dieser Kritikpunkt findet sich deutlich im hier erwähnten Trend wieder.

36 So kann man beispielsweise Microsofts Dokumentation der System.Exception-Klasse die Details der komplexen Struktur der Vererbungsbeziehungen, die Eigenschaften und Methoden des Vererbungsobjekts in der Sprache C#, ihre Unterklassen usw. entnehmen.

37 Siehe unsere obigen Bemerkungen zur Vererbung.

Art der Softwarespezifikation usw.), erst einmal stark verringert. Die verbreitete Praxis, Information über die Problemursachen programmatisch in eine Logdatei zu ‹schreiben› – weil dadurch den Softwareentwicklern die Möglichkeiten gegeben wird, Probleme im Programmentwurf, regelmäßige Fehlerursachen usw. zu identifizieren –, mildert die Unkenntnis der Fehlerquellen zu einem gewissen Maße. Trotzdem muss man sich klarmachen, dass die dadurch gewonnen Informationen die Begriffe voraussetzen, in denen die Software das Problem ursprünglich definiert hat. Im Ergebnis kann man nur Vermutungen über die zugrundeliegenden Ursachen des Problems anstellen (eine Logdatei einer Webanwendung, in die wiederholt Informationen geschrieben werden, dass ein entfernter Datenbankserver nicht antwortet, kann uns nicht sagen, ob dieser Server ausgeschaltet wurde, defekt ist usw.).

Der Punkt ist, dass die Ausnahmebehandlung, indem sie die störungsfreie Ausführung von Software befördert, nicht nur dazu beiträgt, die Software selbst, sondern auch die für ihre Entstehung verantwortlichen Programmierpraktiken zu stabilisieren. Ausnahmen arbeiten daran mit, ein Verständnis von technischen Problemen *als* technischen Problemen aufrechtzuerhalten, indem sie es – normalerweise – erlauben, Fehler als Probleme zu definieren, die der Nutzer erzeugt, weil er die Software nicht versteht (anstatt andersherum). So gesehen macht die Ausnahmebehandlung eine genauere Betrachtung der Beziehung zwischen rechnerischen Objekten und deren Umwelt überflüssig. Obwohl eine solche Stabilisierung der Software erlaubt, eine gewisse Unaufdringlichkeit zu erlangen, führt diese ‹weiche› Eigenschaft dazu, dass es schwieriger wird, ein genaueres Gefühl für die Unterschiede zu bekommen, die durch ihre abstrakte Materialität erzeugt werden.[38]

Fazit: Ontologische Modellierung und der Fall des Unbekannten

Im Verlauf dieses Aufsatzes haben wir versucht, einige Gründe für ein Verständnis von objektorientierter Programmierung zu umreißen, das das Rechnen weniger aus einer epistemischen als aus einer ontologischen Perspektive heraus betrachtet. Es ist wahr, dass es eine historisch gut fundierte Verbindung zwischen Rechnen und Wissensdiskursen gibt, dass Computerprogrammierung die Realität zu modellieren versucht, dass es Verbindungen zwischen Programmiersprachen und formaler Logik gibt usw. Jedoch reicht es nicht aus, die Computerprogrammierung wie eine Wissenschaft im Sinne etwa der Physik, der Chemie oder sogar der Sozialwissenschaften zu verstehen. Im Gegenteil haben wir in diesem Aufsatz – durch die Untersuchung einiger Merkmale der objektorientierten Programmierung – versucht, darauf hinzudeuten, dass die abstrahierende Erfassung und Veränderung von Handlungsmacht durch Software im Kalkül der rechnerischen Objekte besser als ein Ensemble von Techniken zu verstehen wäre, die an der Ausübung *ontologischer Modellierung* beteiligt sind. Mit anderen Worten schließt die Computerprogrammierung ein kreatives

38 Dass die Methoden und Technologien der Softwarekultur mit einer Logik des ‹Gleichen› verbunden sind, die ein tieferes Verständnis der durch sie erreichten Transformationen ausschließt, findet hier einige Elemente einer Erklärung. Über die Logik des Gleichen und Technologie, siehe Isabelle Stengers, *La vierge et le neutrino, les scientifique dans les tournant*, Paris (Empecheurs de Penser en Rond) 2006. Zur Unaufdringlichkeit der Software, siehe Rob Kitchin, Martin Dodge, *Code/Space. Software and Everyday Life*, Cambridge (MIT Press) 2011.

Arbeiten mit den Eigenschaften, Kapazitäten und Tendenzen ein, die von ihrer Umgebung bereitgestellt werden, das auf merkwürdige Weise neue Arten von Entitäten erzeugt, über die sie selbst nur wenig weiß. Solche Entitäten bilden die Struktur für das, was wir ‹abstrakte Materialität› genannt haben – ein Begriff, den wir benutzt haben, um auf die Konsistenz und Autonomie der Zonen oder Territorien hinzuweisen, in denen rechnerische Objekte sich mit anderen Arten von Entitäten verknüpfen.

Ungeachtet des Zusammenhangs, der üblicherweise zwischen Computern und Wissen hergestellt wird – ein Zusammenhang, der in Diskussionen zur objektorientierten Programmierung und der dazugehörigen Modellierung besonders offensichtlich ist –, wird eine gewisse Art von Unerkennbarkeit durch den technisch gestützten Prozess der objektorientierten Abstraktion erzeugt. Dies geschieht nicht einfach nur dadurch, dass die Datenkapselung zu Blackboxen[39] führt, wodurch Wissen und Aktionsmöglichkeiten verpackt und separiert werden –, selbst für diejenigen, die in der Lage wären, mit dem Innenleben solcher Objekte etwas anzufangen –, sondern auch weil die epistemischen Valenzen der Objektorientierung dazu neigen, einen Aspekt der EDV zu verschleiern, der sehr oft mit dem Bedeutungsumlauf *(circulation of meaning)* und der Bedeutungsverfeinerung assoziiert wird, – den der Manipulation und Auswertung von Symbolen. Die Objektorientierung verwendet Interpretation via Abstraktion, um Dinge in der Welt zu erschaffen. Das erinnert an ihre Genese aus zwei Spielarten des Konstruktivismus: des psychologisch abgeleiteten Ansatzes von Kay und anderen am PARC einerseits und des technosozialen Ansatzes aus Skandinavien andererseits. Dass die Leistung von Objekten sich in anderen Arten von Verbindungen an anderen Orten findet und dabei ihre erworbene Fähigkeit hervorhebt, Beziehungen zu verstecken, statt sie aufzudecken, zeigt, wie exemplarisch ihre paradoxe doppelte Handlungsmacht ist.

Man könnte das Thema der Objektorientierung aus vielen Perspektiven behandeln: die hier verwendete Sichtweise insistiert – analog zu Michel Foucaults Diskussion über Macht – darauf, dass das Problem nicht darin besteht, dass die soziotechnischen Praktiken der Programmierung nicht wüssten, was sie tun. Stattdessen führen die Techniken und Technologien der Objektorientierung zu einer Situation, in der man nicht weiß, was man tut.

Aus dem Englischen von Heinz-Günter Kuper, Agata Królikowska und Jens-Martin Loebel

39 Anm. der Übers.: Die Datenkapselung und damit gleichzeitig vertragsähnlich festgelegte Schnittstellenbeschreibung eines Objekts kann durchaus als etwas Positives verstanden werden. Es wird sichergestellt, dass ein Entwickler erwarten kann, dass die Schnittstelle des Objekts konstant bleibt, d. h. dass die Eingabeparameter sich in Form und Anzahl sowie die zurückgelieferten Werte sich vom Datentyp nicht ändern. Eine Schnittstelle wird dadurch zuverlässig. Der Entwickler einer Schnittstelle kann das Innenleben des Objektes im Laufe der Zeit optimieren (z. B. mithilfe von Erweiterungen der Programmiersprache), ohne die Funktion der Schnittstelle zu beeinträchtigen. In der Regel hat der Entwickler eines Objektes und dessen Schnittstelle einen tieferen Einblick in die Wechselwirkungen der Programmteile als ein externer Experte oder Entwickler, der evtl. nur kursorisch diese bestimmte Schnittstelle isoliert betrachtet.