Kontext Architektur – Grundbegriffe zwischen Kunst, Wissenschaft und Technologie

Die Digitalisierung hat den Diskurs der Architektur verändert: Dieser wird mittlerweile von einer Fülle neuer Begriffe bestimmt, die bislang entweder keine oder andere Bedeutungen im Kontext der Architekturtheorie und des architektonischen Entwurfs belegten. Seine Begrifflichkeiten und Strategien werden zunehmend durch Einflüsse geprägt, die an der Schnittstelle zu wissenschaftlichen und kulturellen Vorstellungen der modernen Informationstechnologie entstehen. Vor diesem Hintergrund stellt sich die Frage: Mit welchen praktischen und vor allem auch theoretischen Konzepten kann sich die Architektur mit diesen neuen Technologien auseinandersetzen und in einen fruchtbaren, aber ebenso kritischen Dialog treten? *Kontext Architektur* stellt eine Auswahl jener Begriffe zur Debatte, die im aktuellen Diskurs eine zentrale Rolle spielen. *Kontext Architektur* ist eine Kooperation der Zürcher Hochschule der Künste ZHdK und der Professur Hovestadt für Computer-Aided Architectural Design, ETH Zürich.

In der Reihe Kontext Architektur sind bisher erschienen:

Simulation. Präsentationstechnik und Erkenntnisinstrument
ISBN 978-3-7643-8685-6

Komplexität. Entwurfsstrategie und Weltbild
ISBN 978-3-7643-8687-0

Muster. Ornament, Struktur und Verhalten
ISBN 978-3-7643-8953-6

Code. Zwischen Operation und Narration
ISBN 978-3-0346-0116-0

Kontext Architektur
Eine Zusammenarbeit der ZHdK und der ETH Zürich

Z hdk
Zürcher Hochschule der Künste
Zurich University of the Arts

CAAD Professur Hovestadt
ETH Zürich

Code

Zwischen Operation und Narration

KONTEXT ARCHITEKTUR
Herausgegeben von Andrea Gleiniger und Georg Vrachliotis

Birkhäuser
Basel

Umschlag- und Layoutkonzept: Bringolf Irion Vögeli GmbH, Zürich
Redaktion: Karoline Mueller-Stahl, Leipzig
Reproduktion und Satz: weissRaum visuelle Gestaltung, Basel

Dieses Buch ist auch in englischer Sprache erschienen: *Code. Between Operation and Narration*
ISBN 978-3-0346-0117-7.

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Dieses Werk ist urheberrechtlich geschützt. Die dadurch begründeten Rechte, insbesondere die der Übersetzung, des Nachdrucks, des Vortrags, der Entnahme von Abbildungen und Tabellen, der Funksendung, der Mikroverfilmung oder der Vervielfältigung auf anderen Wegen und der Speicherung in Datenverarbeitungsanlagen, bleiben, auch bei nur auszugsweiser Verwertung, vorbehalten. Eine Vervielfältigung dieses Werkes oder von Teilen dieses Werkes ist auch im Einzelfall nur in den Grenzen der gesetzlichen Bestimmungen des Urheberrechtsgesetzes in der jeweils geltenden Fassung zulässig. Sie ist grundsätzlich vergütungspflichtig. Zuwiderhandlungen unterliegen den Strafbestimmungen des Urheberrechts.

© 2010 Birkhäuser GmbH
Basel
Postfach 133, CH-4010 Basel, Schweiz

Gedruckt auf säurefreiem Papier, hergestellt aus chlorfrei gebleichtem Zellstoff. TCF ∞

Printed in Germany

ISBN 978-3-0346-0116-0

9 8 7 6 5 4 3 2 1

www.birkhauser-architecture.com

7 Andrea Gleiniger & Georg Vrachliotis
EDITORIAL

13 Karim Bschir
VORLÄUFIG ODER GRUNDLEGEND? – DIE CODE-METAPHER IN DER MOLEKULARBIOLOGIE

29 Gabriele Gramelsberger
STORY TELLING WITH CODE

41 Georg Trogemann
CODE UND MASCHINE

55 Claus Dreyer
ARCHITEKTONISCHE CODES AUS SEMIOTISCHER SICHT

75 Georg Vrachliotis
GROPIUS' FRAGE ODER ÜBER DAS OFFENLEGEN UND VERDECKEN VON CODE IN ARCHITEKTUR UND KUNST

91 Andrea Gleiniger
STIL ODER CODE – VON DEN PARADIGMEN DES ARCHITEKTONISCHEN AUSDRUCKS IN DER GEGENWART

Appendix:
109 Ausgewählte Literatur
117 Abbildungsnachweis
119 Biografien der Autorinnen und Autoren

Georg Trogemann

CODE UND MASCHINE

Codes sind in der Informatik eng verbunden mit dem Begriffsdreieck *Kommunikation – Nachricht – Information.* Gemeinsame Basis dieser Begriffe ist die semiotische Betrachtung von Kommunikationsabläufen, die davon ausgeht, dass alle Kommunikation als Sendung von Nachrichten aufgefasst werden kann. Jede Darstellungsform einer Nachricht ist in diesem Zusammenhang eine «Codierung» und jedes System von Zeichen – das durch vorherige Übereinkunft dazu bestimmt ist, die Information zu repräsentieren und sie zwischen Sender und Empfänger zu übertragen – heißt «Code». Im engeren mathematischen Sinn ist ein Code eine Vorschrift für die eindeutige Zuordnung der Zeichen eines Zeichenvorrats zu denjenigen eines anderen Zeichenvorrats. Ziel der Codierungstheorie innerhalb der Informatik ist es, Methoden bereitzustellen, um Nachrichten unterschiedlichster Herkunft mit geeigneten Zeichensystemen möglichst *schnell, fehlerfrei* und *effizient* zu speichern und zu übertragen. Der Begriff «Code» wird dabei in doppelter Weise verwendet, einmal für die mathematische Abbildungsvorschrift, zum anderen für die durch die Codierung entstehenden Zeichenketten.

Die innere Struktur der Zeichen, also die Beziehung der Zeichen zueinander, wie auch die Abbildungsvorschriften zwischen Zeichenvorräten sollen nicht Gegenstand der hier angestellten Überlegungen sein. Wir wollen nicht einzelne Codierverfahren oder formale Sprachen und ihre Vor- und Nachteile betrachten, sondern die Frage stellen, wie Codes innerhalb der Gesamtkonfiguration gegenwärtiger Rechnerarchitekturen ihre Wirkmacht entfalten. In der Computerpraxis vor 1970 war einzig die teure Hardware von wirtschaftlichem Interesse und Programme wurden als kostenlose Zugabe betrachtet. Heute haben sich die Verhältnisse umgekehrt. Die Welt des Rechnens ist bestimmt von *flottierenden Codes,* die sich frei durch die Netze bewegen und die billige Hardware nur noch als wählbare Umgebung betrachten. Beim *cloud computing,* der kommenden Stufe der Rechnervernetzung, werden wir nicht mehr wissen, wo die Programmcodes gerechnet und wo die Daten gespeichert werden. Das weltumspannende Netz prägt bereits heute nicht nur die globale Ökonomie, sondern zunehmend auch unsere Wahrnehmung und Erkenntnis, ja die Wirklichkeit insgesamt scheint durch Computer hindurch konstruiert zu werden. Die Codes im Sinne von Quellcode oder Programmcode, das heißt die nach den Regeln konkreter Programmiersprachen abgefassten *Algorithmen,* spielen hierbei eine entscheidende Rolle.

Code und Erkenntnis

«Am Ende verfällt alles dem algorithmischen Zugriff.»[1] *Frieder Nake*

Programme avancieren zu unverzichtbaren Assistenten, die maßgeblich an der Produktion neuer Erkenntnisse und Methoden beteiligt sind, zum Beispiel bei der Erfindung neuer Werkstoffe, bei komplexen Beweisführungen in der Mathematik oder bei der Simulation des Klimas. Aber nicht nur die Natur- und Ingenieurswissenschaften, auch klassische Entwurfsdisziplinen wie Design und Architektur stützen ihre Arbeit auf die verschiedensten Software-Werkzeuge. Wie kommt es zu dieser Allgegenwart der Codes und welchen versteckten Einfluss haben sie auf die Ergebnisse der Arbeitsprozesse?

Mit der durchgreifenden Axiomatisierung und Formalisierung in der Mathematik und ihrer konsequenten Anwendung in den Naturwissenschaften ging unvermeidlich die Verselbständigung der Zeichen einher. Bis zu David Hilbert hatten die Zeichen noch einen – wenn auch bereits gelockerten – Bezug zur Welt gehabt. Während die Euklidische Geometrie noch Definitionen des Punktes und der Geraden versucht hatte, ging der Göttinger Mathematiker David Hilbert in seiner Axiomatik der Geometrie ganz formal vor. Einzig und alleine die Verknüpfungen waren für ihn von Interesse, die Objekte selbst blieben undefiniert und wurden nur noch gewohnheitsmäßig mit Worten aus dem Sprachgebrauch der Geometrie belegt. Hilbert selbst forderte: Man muss jederzeit anstelle von «Punkt», «Gerade» oder «Ebene» auch «Tisch», «Stuhl» und «Bierseidel» sagen können. Mit seinem strikt formalen Ansatz wollte er die gesamte Mathematik auf der Grundlage eines konsistenten Axiomsystems aufbauen und die Zweifel an der Sicherheit mathematischer Schlussfolgerungen ein für alle Mal aus der Welt schaffen. In Hilberts Programm waren die Zeichen noch mit den Wahrheiten in der Welt verbunden. Die Postmoderne verzichtete dagegen vollständig auf den Wahrheitsanspruch semiotischer Repräsentationen. Codes bildeten nun eine eigene Realität, sie standen nicht mehr für die Welt, sondern für sich selbst. Sie wurden zu Platzhaltern für beliebige Inhalte, und der Umgang mit ihnen beschränkte sich auf ein schematisches Operieren mit Zeichen. Beziehungen zwischen den Zeichen und den beobachtbaren Phänomenen beanspruchten keine Allgemeingültigkeit mehr, sondern unterlagen ausschließlich der Zweck-

1 Aus dem Vortrag «WolkenBilder, wolkig», gehalten auf der *HyperKult 18* in Lüneburg am 4. Juli 2009.

mäßigkeit – Pragmatismus statt Wahrheit. Doch diese Einschränkung des Wahrheitsanspruchs ist nur die eine Seite der Medaille, unterschätzt wird die Realitätsbildung, die von den Zeichen selbst ausgeht.

Obwohl formale Systeme in der Mathematik gründen, wirken sie auf einer Ebene, auf der sich die Mitglieder einer Gemeinschaft, trotz all ihrer Verschiedenheit, aufeinander beziehen können. Jedem denkenden Subjekt ist diese allgemeine Rationalität zugänglich. Rationales Denken ist, sobald es von hinreichend Vielen angewandt wird, immer auch öffentliches und gesellschaftliches Denken. Voraussetzung für die Teilnahme an dieser Form des öffentlichen Denkens ist Abstraktionsfähigkeit, sie ist die individuelle Basis. Rationales Wissen, das in codierter Form vorliegt, wird insofern öffentliches und objektiviertes Wissen, als es von jedem nachvollzogen werden kann, der in der Lage ist, die formalen Codes zu lesen. Rationale Prozesse, die in Maschinen ausgelagert werden, sind in dieser Sichtweise eine Form von objektivierter Handlungstheorie. Holling und Kempin[2] bezeichnen sie als «implementierte Theorie», wobei sie unter einer «Implementierung» generell die Umsetzung eines formalen Verfahrens in eine Maschine verstehen. Damit sind implementierbare Theorien eine Teilmenge der möglichen Theorien, die sich dadurch auszeichnen, dass sie *operationalisierbar* sind. Die beiden Autoren weisen an gleicher Stelle darauf hin, dass die Unterschätzung des Formalen eine lange Tradition hat, und dies nicht nur in den Geisteswissenschaften, die Formalismen für ein begrenztes Hilfsmittel der Naturwissenschaften halten, sondern auch in der Mathematik selbst, die ihren Zeichensystemen ebenfalls keinerlei gesellschaftliche Bedeutung beimisst. Die eigentliche Bedeutung des Formalen als gesellschaftliche Integrationsinstanz und als implementierte Theorie, wird von beiden Seiten übersehen. Mit der Verbreitung von Computern ist auch die Verbreitung der formalen Codes noch einmal explosionsartig angewachsen, dennoch bleiben sie weiterhin nahezu unsichtbar und unbeachtet.

Wir wollen etwas genauer auf die Frage eingehen, wie es überhaupt möglich ist, Denkprozesse auf Maschinen zu übertragen. Die Antwort lautet: Der Denkprozess muss lediglich in einen maschinellen Handlungsprozess übersetzt werden. Hierbei kommt der Formalisierung die entscheidende Vermittlerrolle zu. Formalismen dienen der exakten symbolischen Beschreibung von Methoden,

2 Eggert Holling, Peter Kempin: *Identität, Geist und Maschine – Auf dem Weg zur technologischen Gesellschaft*, Hamburg 1989, S. 82.

wobei Methoden unter dem Gesichtspunkt der Formalisierung nichts anderes sind, als Regeln für unser Handeln, die es gestatten, aus unendlich vielen Handlungsalternativen eine bestimmte auszuwählen. Bereits geleistete Verständnisarbeit wird mit Hilfe formaler Strukturen funktionalisiert und damit zu erlernbarem Erkenntnisbesitz und zum übertragbaren Verfahren. «Formalisierung ist nichts anderes als die handlichste Art solcher Funktionalisierung des einmal Geleisteten; aber sie ist eben auch potenziell schon Technisierung, denn was formalisiert werden kann – das heißt: was seine Anwendbarkeit unabhängig von der Einsichtigkeit des Vollzuges gewinnt –, das ist im Grunde auch schon mechanisiert, auch wenn die realen Mechanismen zu seiner Speicherung und geregelten Assoziation nicht bereit gestanden haben. Alle Methodik will unreflektierte Wiederholbarkeit schaffen, ein wachsendes Fundament von Voraussetzungen, das zwar immer mit im Spiele ist, aber nicht immer aktualisiert werden muss.»[3] Wir erleben derzeit, wie die Algorithmisierung unserer Lebenswelt immer weiter ausgreift und – im Diktum Frieder Nakes – am Ende alles dem algorithmischen Zugriff verfällt. Die eigentliche Bedeutung der Codes liegt dabei in ihrer Funktion als gesellschaftliche Integrationsinstanz. Was codiert, auf tausende Maschinen übertragen und durch permanente Widerholung Teil unserer Alltagswelt wird, stabilisiert sich selbst und wird schließlich zum kulturellen, unhinterfragten Sediment. Als implementierte und *lauffähige Theorie,* gehen die formalen Codes dabei mit der Maschine eine neuartige und mächtige Verbindung ein. Computer sind nicht nur passive Träger von Zeichen, sondern aktive Erzeuger – Zeichen erzeugen Zeichen. Doch wäre es ein Fehler zu glauben, wir bekämen genau jene Phänomene zurück, die wir in der Entwurfsphase der Programme schon hineingesteckt haben. Der Abstraktionsprozess, der notwendig ist, um zu Algorithmen und operierenden Zeichen zu kommen, läuft während der Ausführung der Codes zwar gewissermaßen rückwärts ab, aber nicht identisch. Wenn Abstraktion ein Zurücktreten, Generalisieren und Reinigen der Phänomene von Unwesentlichem und Zweideutigem ist, dann laden die analogen Interfaces in der Programmausführung die Ergebnisse wieder mit Unbeabsichtigtem, Unschärfen und Mehrdeutigkeiten auf, zum Beispiel beim Übergang vom bildgenerierenden Algorithmus zum Bild selbst. Aber aufgeladen wird nun natürlich Anderes als das, was vorher durch Abstraktion, also auf dem Weg zum Algorithmus, weggeworfen wurde.

3 Hans Blumenberg: *Wirklichkeiten, in denen wir leben,* Ditzingen 1981, S. 41.

Universalität durch Codierung

«Die Bedeutung der universalen Maschine ist klar. Wir brauchen nicht unzählige unterschiedliche Maschinen für unterschiedliche Aufgaben. Eine einzige wird genügen. Das technische Problem der Herstellung verschiedener Maschinen für verschiedene Zwecke ist ersetzt durch die Schreibarbeit, die Universalmaschine für diese Aufgabe zu programmieren.»[4] *Alan Mathison Turing*

In seinem berühmten Aufsatz «On Computable Numbers, with an Application to the Entscheidungsproblem» beschreibt Alan Turing das Prinzip der universellen Maschine wie folgt: «Es ist möglich, eine einzige Maschine zu erfinden, die dazu verwendet werden kann, jede berechenbare Folge zu errechnen. Wenn diese Maschine *U* mit einem Band gespeist wird, dessen Anfang mit der D.N.[5] irgendeiner rechnenden Maschine *M* beschriftet ist, so wird *U* dieselbe Funktion wie *M* errechnen.»[6] Praktisch umgesetzt heißt das, man codiert die Liste von Instruktionen für eine spezielle Turingmaschine *M* und schreibt diese auf den vorderen Teil des Bandes. Codieren heißt, dass wir im Falle einer binären universellen Maschine nicht nur die Daten, sondern auch das Programm als Folge von Nullen und Einsen darstellen. Dieser Bandabschnitt ist nun der erste Teil des Inputs für die universelle Turingmaschine *U*. Sie bearbeitet den restlichen Teil des Bandes, indem sie die Instruktionen auf dem vorderen Teil des Bandes ausführt und so die spezielle Maschine *M* exakt imitiert. Heutige programmierbare Computer funktionieren im Prinzip genauso. Sie simulieren das Verhalten einer speziellen Maschine dadurch, dass der Programmcode zusammen mit den Daten im Speicher der Maschine steht. Die Universalmaschine ist ohne Software zwar offen für alle denkbaren Verhaltensweisen, für sich alleine ist sie allerdings handlungsunfähig. Erst bei Programmstart verhält sich die Universalmaschine dann wie die spezielle Maschine, die durch das Programm definiert ist.

Universelle Rechner bestehen damit aus zwei maschinellen Schichten:
1. einer Hardware, die an die Gesetze des klassischen Materials gebunden ist,
2. den Programmcodes, die nur noch algorithmischen Gesetzen unterliegen.

4 Alan Mathison Turing: *Intelligente Maschinen,* zitiert nach: Bernhard Dotzler, Friedrich Kittler (Hrsg.): *Alan Turing, Intelligence Service. Ausgewählte Schriften,* Berlin 1987, S. 88.
5 D.N. = *description number* (Beschreibungszahl).
6 Ebenda, Beitrag: «Über berechenbare Zahlen mit einer Anwendung auf das Entscheidungsproblem», S. 31.

Die erste Schicht – die Hardware – ist naturgebunden und gehorcht den Gesetzen unserer physikalischen Welt. Sie verbraucht Energie, unterliegt Alterungsprozessen, die irgendwann zu Fehlfunktionen führen, sie kann durch einwirkende Kräfte (mechanische Stöße, Magnetismus, Überspannung etc.) beschädigt werden. Diese materielle Maschine kann auf der Basis physikalischer Prinzipien logische Elementaroperationen realisieren. Auf dieser elektronischen (mechanischen, optischen etc.) Maschine sitzt eine zweite Schicht, die den physikalischen Gesetzen nicht mehr unterliegt, sondern nur noch der menschlichen Rationalität und den Grenzen des Formalen verpflichtet ist. Diese codebasierte Maschine kennt keine materiellen Schwächen mehr, sondern nur Ausfälle, die auf logische Fehler des Programmierers zurückgehen. Die codebasierte Maschine realisiert die Welt der Algorithmen, sie steuert das Verhalten der unteren Schicht und ist dabei strikt an formale Sprachen und an die Gesetze der menschlichen Logik gebunden. Auf dieser algorithmischen Ebene der Maschine lassen sich zum Beispiel neue physikalische Welten beschreiben und simulieren, in der ganz neue und andersartige Naturgesetze gelten. Wir können unsere Reflexion über die physikalische Welt in Form von Algorithmen an die Maschine abgeben, diesem logischen Konstrukt in der Simulation aber auf vollkommen neue Weise wieder begegnen, nicht mehr als formallogische Beschreibung, sondern als Erfahrungswelt. Die Grenzen dieser zweiten Maschine liegen also nicht in unserer physikalischen Welt, sondern den Begrenzungen der Codes und dem menschlichen Vermögen, algorithmische Welten zu erfinden.

Reflexivität der Codes

In der programmierbaren Maschine werden sowohl Daten als auch Programme auf der Grundlage identischer Zeichen codiert. Die Zeichen können deshalb, anders als in der Arithmetik, welche strikt zwischen *Operanden* und *Operatoren* trennt, in neue Verhältnisse zueinander treten. Im Rechner können nicht nur Daten maschinell verarbeitet werden, sondern auch Programme. Während in der Architektur Gebäude von Architekten entworfen werden, sind es in der Architektur der Rechner vor allem andere Rechner, die aktiv die nächste Generation von Rechnern mitentwickeln. Möglich ist das durch das Umschlagen der Codes. Wir können Codes als Daten behandeln aber auch aktiv manipulieren, das bedeutet, im nächsten Moment können diese selbst zu rechnenden Einheiten werden. Jedes Herunterladen und starten eines Programms aus dem Netz ist ein Beispiel

für diesen Wechsel zwischen *Code als Operand* und *Code als Operator.* Während des Transfers durch das Netz ist der Programmcode passiv und wird von der Netzwerksoftware selbst als Daten behandelt. Durch das Starten des Programms auf dem Rechner schlägt der Code vom Status des Operanden um in den Modus des Operators. Als Operator wird der Code nun selbst Daten manipulieren. Erst durch die Reflexivität des Codes erhält die Maschine ihre praktische Bedeutung. Ohne selbstbezügliche Verarbeitungsmechanismen wäre die Arbeit am Rechner bis heute ein Umgang mit Nullen und Einsen. Zur Verdeutlichung der reflexiven Beziehungen, die zwischen den *Codes als Operanden* und den *Codes als Operatoren* herstellbar sind, wollen wir zwei etwas komplexere Beispiele anführen: die *Compiler* und die Genetische Programmierung.

Compiler sind Programme, die als Eingabe Zeichen erhalten (ein Programm *a*) und daraus neue Zeichen (Programm *a'*) erzeugen. Wichtig an der Funktion des *Compilers* ist, dass er seine Arbeit im ersten Schritt so ausführt, dass im zweiten Schritt ein Umschlag des Programms vom Status des Operanden in den Status des Operators stattfinden kann. *Compiler* müssen, um von der Maschine ausgeführt werden zu können, als Maschinenprogramme vorliegen. Ihre Erstellung ist eine sehr aufwendige Entwicklungsaufgabe. Da sich jeder *Compiler* auf eine bestimmte Programmiersprache und eine bestimmte Maschinensprache bezieht, müssen für *n* Programmiersprachen und *m* Maschinensprachen *n x m Compiler* geschrieben werden, was einen mit der Anzahl *n* und *m* quadratisch wachsenden Aufwand bedeutet. Durch geschickte Ausnutzung von Code-Reflexivitäten besteht auch hier die Möglichkeit, *Compiler* bis zu einem gewissen Grad durch sich selbst zu übersetzen. Das unter dem Begriff *«bootstrapping»* in der Compilerentwicklung schon sehr früh verwendete Verfahren, erstellt bei Vorliegen eines *Compilers a* für eine bestimmte Implementierungssprache und eine Rechenanlage *A* mit geringem Aufwand einen *Compiler b* für diese Implementierungssprache und eine Maschine *B*.

1. Im *Compiler a*, der als Code in der Implementierungssprache vorliegt, werden an allen Stellen, an denen Maschinensprache für die Maschine *A* erzeugt wird, die entsprechenden Befehle der Maschine *B* eingesetzt. Durch diesen Vorgang erhält man *a'*.
2. Man übersetzt *a'* mit Hilfe von *A* und *a* in die Maschinensprache von *A* und erhält *a''*.
3. *a''* ist ein lauffähiges Programm für die Rechenanlage *A*. Dieses Programm erhält als Input wiederum *a'* und erzeugt ein Maschinenprogramm *b*, das auf der

Rechenanlage *B* lauffähig ist. *B* ist der gewünschte *Compiler,* der auf *B* lauffähig ist und Programme erzeugt, die ebenfalls auf *b* laufen.

Die *Genetische Programmierung* gehört zur Klasse der *Evolutionären Algorithmen,* die als heuristische Optimierungsverfahren auch dann noch gute Ergebnisse für eine Problemstellung liefern, wenn geschlossene Lösungen nicht zur Verfügung stehen. Sie arbeiten nach dem Vorbild der biologischen Evolution auf Basis von Populationen von Individuen, die von Generation zu Generation ihren genetischen Code vererben. Die Eigenschaften der einzelnen Individuen sind als Genotypen codiert. Sie repräsentieren das Erbgut der Individuen und liefern Baupläne für die Individuen. Die Reproduktion einer Population von einer Generation in die nächste erfolgt durch identisches Kopieren, die Mutation einzelner zufällig gewählter Positionen des Codes sowie den Austausch von Teilstrukturen zwischen zwei Individuen *(crossover).* Veränderungen der Merkmale eines Individuums können ausschließlich am Genotyp (den Zeichenketten) vorgenommen werden, die Selektion der Besten gemäß Fitness-Funktion erfolgt dagegen am Phänotyp. Der Phänotyp ist die physikalische Manifestation des Genotyps, also die Merkmalsausprägung eines Individuums in seiner Umwelt. In der *Genetischen Programmierung* werden spezielle Genotypen behandelt, nämlich Computerprogramme. Der Genotyp ist hier der Programmcode und der Phänotyp die Performance des Programms, wenn es auf einem Rechner ausgeführt wird. Auch hier erkennen wir wieder den Mechanismus des Umschlagens zwischen *Code als Operator* und *Code als Operand.* Für den genetischen Basisalgorithmus fungiert der Genotyp als Operand, der Phänotyp dieses Codes dagegen als Operator.

Maschinelle Selbstreproduktion

Das skizzierte Prinzip der *Genetischen Programmierung* ist gleichzeitig ein Beispiel für die oben beschriebene Aufspaltung des Universalrechners in zwei Maschinen. Mittels evolutionärer Algorithmen erhalten wir Codes, die in der Lage sind, sich selbst zu manipulieren und in ihrer Performanz weiterzuentwickeln. Die untere Schicht der Maschine, die Hardware, auf der das genetische Programm läuft, bleibt davon vollkommen unberührt. Im Folgenden wollen wir zeigen, dass die Ausweitung der Selbstmanipulationsstrategien von Codes auf die physikalische Welt in der klassischen Variante ebenfalls Codes verwendet.

Hierzu müssen wir allerdings eine scharfe Unterscheidung zwischen «Selbstreplikation» und «Selbstreproduktion» treffen. «Selbstreplikation» ist ein

Prozess, durch den ein Objekt eine Kopie von sich selbst herstellt. Im Falle maschineller Replikation lässt sich dies als deterministischer Prozess realisieren, bei dem es vor allem darauf ankommt, dass keine Fehler passieren. Codes, die als explizite Beschreibungen der Maschine vorliegen, sind dafür nicht erforderlich. «Selbstreproduktion» enthält im Unterschied dazu eine Aufwärtsbewegung zur Vielfalt, sie implementiert einen Entwicklungsprozess, bei dem Abweichungen und Unterschiede zur Vorgängergeneration nicht nur zulässig sind, sondern erwünscht. Die Selbstreproduktion ist damit ein selbsterhaltendes System, das dem Darwinschen Evolutionsgedanken folgt. Für die Simulation des Lebens würde die Selbstreplikation nicht genügen, weil sie keine vererbbaren Mutationen kennt. Um maschinelle Selbstreproduktion zu realisieren, scheint es aber vorteilhaft zu sein, dass Maschinen nicht nur Material ihrer Umgebung verarbeiten, um daraus Kopien ihrer Selbst zu bauen, sondern gleichzeitig mit expliziten Informationen umgehen, die als Codes vorliegen. Dabei handelt es sich nicht um irgendwelche Codes, sondern um reflexive Informationen, also Beschreibungen ihrer eigenen Konstruktion. Der erste Prozess dieser Art, der nicht nur die einfache Selbstreplikation erlaubt, sondern das evolutionäre Problem der Selbstreproduktion löst, geht auf den Mathematiker John von Neumann zurück.

Auf den ersten Blick erscheint diese Fähigkeit wie ein *Circulus vitiosus,* da wir erwarten würden, dass die Kompliziertheit von Systemen, die andere Systeme bauen, von den *Eltern* zu den Nachkommen abnimmt. Damit ein Automat *A* einen Automaten *B* bauen kann, muss er schließlich nicht nur eine vollständige Beschreibung von *B* enthalten, sondern auch noch verschiedene Vorrichtungen besitzen, um die Beschreibung interpretieren und die Bauarbeit ausführen zu können. Die plausibel erscheinende Annahme, die Kompliziertheit von sich selbst bauenden Automaten müsste von Generation zu Generation abnehmen, steht aber im Widerspruch zur offensichtlichen Selbsterhaltungsfähigkeit der Natur. Organismen pflanzen sich fort und produzieren neue Organismen, die mindestens genauso kompliziert sind wie sie selbst. Im Laufe langer Evolutionsperioden kann die Kompliziertheit, wie wir wissen, sogar zunehmen. Wie aber sehen die allgemeinen logischen Prinzipien aus, die Automaten zur Selbstfortpflanzung befähigen und sogar eine Steigerung an Kompliziertheit ermöglichen? Von Neumanns Schlussfolgerung lautet: Es gibt ein minimales Niveau von Kompliziertheit, auf dem Automaten möglich sind, die sich selbst fortpflanzen oder sogar höhere Gebilde bauen. Unterhalb dieses Niveaus sind Automaten

degenerativ, das heißt Automaten, die andere Automaten bauen, sind nur in der Lage, einen weniger komplizierten zu erzeugen. Insgesamt betrachtete John von Neumann fünf verschiedene Modelle der Selbstreproduktion. Burks nennt sie: das kinematische Modell, das zellulare Modell, das Erregungs-Schwellwert-Ermüdungs-Modell, das kontinuierliche Modell und das probabilistische Modell.[7] Für das Problem der kinematischen Selbstfortpflanzung, das wir im Folgenden näher betrachten wollen, reicht das Konzept der Turingmaschine nicht aus, da sie nur Zeichen auf einem Stück Band produzieren. Von Neumann zielt mit dem kinematischen Modell aber auf Automaten ab, die Automaten bauen, die also nicht nur Zeichen manipulieren, sondern auch Hardware bauen. Das Grundprinzip des kinematischen Systems, über das von Neumann seit mindestens 1948 nachdachte, lässt sich leicht skizzieren.

Von Neumanns selbstreproduzierender Automat ist ein Aggregat aus drei Automaten und ihren jeweiligen Beschreibungen (Bauplänen).[8] Erstens einem *Konstruktionsautomat A*, der in der Lage ist, jeden beliebigen Automaten *X* zu bauen, wenn er mit einer Beschreibung des Automaten *F(X)* gefüttert wird, zweitens einem *Kopierautomat B*, der von jeder Beschreibung eine identische Kopie herstellen kann, und drittens einem *Kontrollautomat C* der das Zusammenspiel der Automaten *A* und *B* kontrolliert. Zunächst aktiviert dieser den Automaten *A*, der sofort beginnt, den Automaten *X* gemäß der Beschreibung *F(X)* zu bauen. Anschließend aktiviert er den Automaten *B*, der daraufhin eine Kopie von *F(X)* herstellt. Diese Kopie setzt *C* in den neuen Automaten *X* ein, der gerade von *A* gebaut wurde. Als letztes trennt *C* die neu entstandene Konstruktion *X + F(X)* vom Konstruktionsautomaten ab.

Wir können nun das Gesamtaggregat *A + B + C* mit *D* bezeichnen und erhalten den gesuchten *Selbstreproduktionsautomat D + F(D)*, wobei *F(D)* wieder eine Beschreibung des Automaten *D* ist. Wichtig am Von-Neumannschen-Prinzip der Selbstreproduktion ist, dass es nicht ohne Codierung auskommt. Die Maschinen müssen den Umweg über die Codierung ihrer eigenen Konstruktion gehen, um reproduktionsfähig zu sein. Doch dadurch repliziert sich der Automat *D + F(D)* nicht einfach nur, sondern besitzt weitere wichtige Eigenschaften. Wir sehen

7 Vgl. John v. Neumann: *Theory of Self-Reproducing Automata*, herausgegeben und vervollständigt von Arthur W. Burks, Urbana 1966.

8 Vgl. Georg Trogemann, Jochen Viehoff: *CodeArt – Eine elementare Einführung in die Programmierung als künstlerische Praktik*, Wien, New York, 2005, S. 443ff.

zum Beispiel, dass zufällige Änderungen an *F(D)* Merkmale aufweisen, die in der Biologie in Verbindung mit Mutationen auftreten. Die meisten willkürlichen Änderungen an *F(D)* werden dazu führen, dass der Automat nicht mehr funktionsfähig ist. Bei einer genügend großen Zahl von Änderungen werden jedoch einige dabei sein, die zu neuen funktionsfähigen Automaten führen. Diese neuen Automaten besitzen andere Eigenschaften als ihre Vorgänger.

Kleine Änderungen in der Beschreibung des Automaten erlauben Konstruktionsprozesse, die nicht nur den Automaten reproduzieren, sondern auch noch Nebenprodukte herstellen. Sei *E* ein Automat mit zugehöriger Beschreibung *F(E)*; wenn wir obigen selbstreproduzierenden Automaten *D = (A + B + C)* mit *F(A + B + C + E)* füttern, wird ein neuer Automat *(A + B + C + D) + F(A + B + C + E)* erzeugt. Der neue Automat stellt nicht nur eine Kopie seiner selbst her, sondern produziert auch noch den Automaten *E*.

Dieses Spiel der Automaten benötigt keinen Benutzer mehr, ihre Vervielfältigung und Weiterentwicklung läuft, sobald der Prozess einmal angestoßen ist, ohne weiteres Zutun ab. Abgesehen von den Gefahren, die eine vollständige Aufgabe der menschlichen Kontrolle birgt, ist diese Form maschineller Autonomie auch sonst für die meisten Arbeitsprozesse überhaupt nicht erstrebenswert. Viel interessanter sind Werkzeuge, die die Handlungsmöglichkeiten ihrer Benutzer erweitern, nicht die der Maschinen. In diesem Zusammenhang stellt sich dann die Frage, wie weit die Entscheidungen des Programmierers und die deterministisch ablaufenden Codes die erzielbaren Arbeitsergebnisse mitbestimmen. Welche kreativen Spielräume erlauben Softwarewerkzeuge ihren Nutzern?

Die Offenheit der Codes

Ein Vorgang ist dann formal beschreibbar, wenn er die Bedingungen der Schriftlichkeit, der Schematisierbarkeit und der Interpretationsfreiheit erfüllt.[9] Schriftlichkeit bedeutet, dass wir den Vorgang mit Hilfe von Zeichen ausdrücken können. Schematisierbarkeit heißt, der Ablauf lässt sich als festes Schema und allgemeines Verfahren (also algorithmisch) beschreiben, und Interpretationsfreiheit meint, dass wir vollkommen frei in der Wahl der Zeichen und Bezeichnungen sind, da wir bei der Ausführung der Operationen keinen Bezug mehr darauf nehmen, wofür die Zeichen stehen. Durch den Vorgang der Formalisierung entledigen wir uns auf der Ebene der Zeichen jeder Bedeutung. Der

9 Vgl. Sybille Krämer: *Symbolische Maschinen*, Darmstadt 1988, S. 1–4.

Außenstehende kann nur dann die Bedeutung aus den Zeichen lesen, wenn er den Schlüssel der Codierung kennt. Alle drei Bedingungen der Formalisierung sind automatisch erfüllt, wenn es gelingt, einen Ablauf in einer maschinenlesbaren Programmiersprache auszudrücken. Formale Strukturen, so die damit einhergehende Vorstellung, lassen der Imagination keinen Raum. Durch die Formalisierung sollen Gegebenheiten ja gerade möglichst exakt erfasst und jede Missdeutung und Unschärfe ausgeschaltet werden. Es stellt sich die Frage, wo beim Umgang mit dem Computer die Freiräume für die Partizipation und die Imagination des Nutzers liegen. Erlauben die formalen Grundmuster des Computers und seiner Programmierung lediglich starre unkreative Ergebnisse? Wie weit ist der Architekt, der mit Programmen arbeitet, überhaupt noch frei in seinen Entwürfen und Entscheidungen. Sind nicht in jeder Software lange vor der Einzelproduktion alle Entscheidungen bereits im Werkzeug festgelegt, die einstmals im freien Ermessen des Architekten lagen. Gibt es dennoch Spielräume, die freie Entwürfe jenseits der Setzungen durch den Programmierer zulassen?

Unter Offenheit eines Programmcodes wollen wir ganz allgemein die Eigenschaft verstehen, dem Benutzer jeweils neue Handlungs- und Interpretationsspielräume zu eröffnen und in der Arbeit mit dem Programm eine gewisse Flexibilität und Durchlässigkeit für seine Intentionen zu schaffen. Der Programmtext ist aber immer das explizit und unzweideutig Gegebene. Auf dieser Beschreibungsebene können Freiheiten nur erreicht werden, indem wir die bereits oben diskutierten Selbstveränderungsstrategien für Codes einsetzen wie Lernverfahren, Selbstreferenzen oder allgemein Code-Reflexivitäten. Nur indem der Code sich selbst fortschreibt, kann auf der performativen Ebene des Codes wirklich Neues entstehen, das auch aus der Sicht des Programmierers als unvorhergesehen gewertet werden muss. Tatsächlich kann Offenheit einer Applikation aber sehr viel einfacher ohne explizite Änderung des Codes in jeder gewöhnlichen Anwendung erreicht werden. Um Offenheit zu erhalten – so die hier vertretene These – dürfen genau die Qualitäten, die offen gehalten werden sollen, nicht formalisiert werden. Hierzu ein Beispiel aus der Architektur.

Umberto Eco klassifiziert die architektonischen Codes in syntaktische und semantische.[10] Syntaktische Codes bilden dabei das Wissen von den Konstruktionen nach. Hierzu gehören nach Eco zum Beispiel Balken, Decken, Gewölbe, Auflager, Bögen, Pfeiler, Platten, Betontragwerke. Diese syntaktischen Codes besitzen noch keinen Bezug zur Funktion oder zum denotierten Raum, darauf verweisen erst semantischen Codes. Zu den semantischen Kategorien gehören beispielsweise primäre Funktionen wie Dach, Terrasse, Treppe und Fenster aber auch *Wohnideologien* (Gemeinschaftssaal, Tages- und Nachtzonen, Esszimmer, Aufenthaltsraum), *soziale typologische Gattungen* (Krankenhaus, Villa, Schule, Schloss, Bahnhof) und *räumliche typologische Gattungen* (runde oder kreuzförmige Grundrisse, Labyrinthe).

10 Umberto Eco: *Einführung in die Semiotik*, München 1994 (ital. Originalausgabe 1968, dt. Erstausgabe 1972), S. 329.

Eine Architektur, die mit solchen Codes arbeitet, kann nach Eco ihren Nutzern aber nichts liefern, worauf sie nicht schon gefasst wären. «Der Gesichtspunkt, auf den man in all diesen Codifizierungen stößt, ist der, dass sie schon ausgearbeiteten Lösungen eine Form geben. Das heißt, es sind Codifizierungen von Botschaftstypen. [...] Die Codes [...] wären also nichts anderes als ikonische, stilistische oder rhetorische Lexika. Sie bieten keine generativen Möglichkeiten, sondern fertige Schemata, nicht offene Formen, über die man reden könnte, sondern verhärtete Formen, allgemeine Beziehungen unerwarteter Art. [...] Es stimmt nicht, dass einige leere und rein differentielle Formen des architektonischen Bedeutens (Pfeiler und Balken) jede mögliche architektonische Kommunikation erlauben: sie erlauben die Art von architektonischer Kommunikation, an die uns die westliche Kultur gewöhnt hat, nach dem Modell gewisser statischer und dynamischer Kriterien und gewisser Regeln der Euklidischen Geometrie, die auch wenn sie stabiler und gegen Verschleiß widerstandsfähiger zu sein scheinen als andere Regelsysteme, uns zwingen, uns innerhalb einer gewissen Grammatik des Bauens zu bewegen. Zumindest findet man sie codifiziert unter der Bezeichnung Baukonstruktionslehre.»[11]

Vor der Herausforderung, die Eco beschreibt, steht nicht nur die Architektur, sondern jeder Entwickler von Werkzeugen für offene Entwurfsprozesse. Die implementierten syntaktischen und semantischen Codes – die Informatik nennt diese Ontologien – entscheiden bereits über die zu erwartenden Ergebnisse. Wer also Überraschendes in Bezug auf bestimmte Konzepte, Begriffe und Ästhetiken erreichen will, darf diese nicht schon in den Formalismus seines Systems stecken. Wer keine Gebäude aus Balken, Decken, Pfeilern und Wänden bauen will, darf diese nicht als Grundelemente seines Entwurfssystems implementieren.

11 Ebenda, S. 329ff.

Was hier als Trivialität erscheint, wird dennoch in verfügbaren Werkzeugen wenig reflektiert oder gar überwunden. Die Schwierigkeit liegt sicher nicht darin, altbekannte Kategorien und Konzepte aufzugeben, sondern diese auf eine andere Ebene der Abstraktion zu verlagern. Die offen zu haltenden Kategorien müssen durch abstraktere Konzepte ersetzt werden, aus denen sie als Spezialfälle rekonstruiert werden können. Sie bleiben selbst dabei unformalisiert, können im Arbeitsprozess aber erzeugt und reflektiert werden.