Context Architecture: Fundamental Concepts Between Art, Science, and Technology

Digitalization has altered architectural discourse. Today, discussions in architectural theory and design are shaped by many new ideas, including some that previously had no meaning in that context, or else very different ones. Increasingly, the conceptualizations and strategies of architectural discourse are molded by influences emerging along the interface joining scientific and cultural images of modern information technology. Posing itself against this background is the question: on the basis of which practical and in particular which theoretical concepts can architecture come to terms with these new technologies, thereby entering into a simultaneously productive and critical dialog with them? Presented for debate in *Context Architecture* is a selection of such ideas, all of them central to current discourses. *Context Architecture* is a collaboration of the Zurich University of the Arts (ZHdK) and the Ludger Hovestadt, chair for Computer Aided Architectural Design at the ETH Zurich.

Also available in the series Context Architecture:

Simulation. Presentation Technique and Cognitive Method
ISBN 978-3-7643-8686-3

Complexity. Design Strategy and World View
ISBN 978-3-7643-8688-7

Pattern. Ornament, Structure, and Behavior
ISBN 978-3-7643-8954-3

Code. Between Operation and Narration
ISBN 978-3-0346-0117-7

Context Architecture
A collaboration of the Zurich University of the Arts (ZHdK) and the ETH Zurich

Z hdk
Zürcher Hochschule der Künste
Zurich University of the Arts

CAAD Professur Hovestadt
ETH Zürich

Code

Between Operation and Narration

CONTEXT ARCHITECTURE
Edited by Andrea Gleiniger and Georg Vrachliotis

Birkhäuser
Basel

Translation from German into English: Laura Bruce, Berlin
Project management: Karoline Mueller-Stahl, Leipzig

Cover and layout design: Bringolf Irion Vögeli GmbH, Zurich
Reproductions and typesetting: weissRaum visuelle Gestaltung, Basel

This book is also available in German: *Code. Zwischen Operation und Narration*
ISBN 978-3-0346-0116-0.

Library of Congress Control Number: 2010923974

Bibliographic information published by the German National Library
The German National Library lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available on the Internet at http://dnb.d-nb.de.

Printed on acid-free paper produced from chlorine-free pulp. TCF ∞

Printed in Germany

ISBN 978-3-0346-0117-7

9 8 7 6 5 4 3 2 1

www.birkhauser-architecture.com

Georg Trogemann

CODE AND MACHINE

In computer science, codes are related to the triangulum *communication – message – information*. The one common basis for these terms is the semiotic analysis of communication processes, which assumes that all communication can be considered a transmission of messages. In this context, any form of presenting a message is a code and every system of symbols – that is supposed to represent and transmit the information between sender and receiver via a predetermined agreement – is called a "code." In a strictly mathematical context, a code is a directive that allows symbols of a coded set of characters to be classified in a clear and precise manner. The objective of a computer science coding theory is to create the *fastest, most accurate,* and *most efficient* methods to store and transmit messages of different origin using the appropriate symbol systems. The term code is applied here in two ways, first for the mathematical mapping, and second for the chain of characters created by the coding.

This essay will not attempt to outline the inner structure of a symbol, in other words, how symbols relate to each other, or the mapping between coded sets of characters, nor will we observe specific coding processes or formal languages and their advantages and disadvantages. We will instead focus on how codes develop their force within the overall configuration of today's computer architecture. Before the 1970s, the only thing considered economically lucrative in the field of computer practice was the expensive hardware – programs were little more than free supplements. But today almost the opposite is true. The world of computers is now defined by *floating codes* that move freely through networks; the inexpensive hardware is an environment that is purely optional. Cloud computing is the next phase of computer networking. In this paradigm, users need not know where the program codes are calculated or where data is stored. The World Wide Web not only shapes the global economy; it shapes our perception and intellect more and more – even reality as a whole seems computer generated. Codes in terms of source code or program code, meaning algorithms written by the same rules of actual program languages, play the decisive role here.

Code and Perception

"Ultimately, everything will succumb to the power of algorithms."[1] *Frieder Nake*

Programs are becoming inescapable assistants that mainly help to produce new knowledge and methods needed, for instance, to develop new materials, solve complex mathematical problems, or simulate climates. Yet, not only science or engineering depends on different software tools, but also classic creative disciplines such as design and architecture. How did the code grow to become so omnipresent and what hidden influence do they have on the results of the work process?

Symbols became independent as an inevitable consequence of the complete process of axiomatization and formalization in mathematics and its logical application in the natural sciences. Before David Hilbert, symbols had a reference to the world, albeit a loose one. While Euclidian geometry was still attempting to define a point and a straight line, mathematician David Hilbert from Göttingen was following a very formal approach to axiomatics. Hilbert was interested only in the links; the objects themselves remained undefined and were mechanically provided with geometry-related names. Hilbert himself claimed that it should always be possible to say, "table," "chair," and "beer mug" instead of "point," "straight," or "plane." His strictly formal approach was an attempt to base mathematics on a consistent system of axioms, and to rid the world once and for all of any doubt about the certainty of mathematical deduction. In Hilbert's Program symbols were still connected to truths in the world. Yet the postmodernists completely rejected any claim to truth in semiotic representations. Codes now formed their own truths; they no longer represented the world, but themselves. They became placeholders for unknown content, and working with them was limited to a schematic handling of symbols. Relationships between symbols and the observed phenomenon were subjected purely to expedience and no longer required universality. Pragmatism appropriated truth. Yet this aspect of the restriction of claims to truth is only one side of the coin, it underestimates the actual reality generated by the symbols themselves.

1 Quoted from the lecture "WolkenBilder, wolkig," (Cloud pictures, cloudy) given at *HyperKult 18* in Lüneburg on July 4, 2009.

Formal systems are grounded in mathematics, yet they nevertheless function at a level on which members of a society can relate to each other, despite their many differences. Every thinking subject has access to this general rationality. If enough people think in a rational way, it will inevitably become public and social thinking. The only requirement needed for this form of public thinking is the ability to abstract, which is the individual starting point. In this respect, any rational knowledge available in coded form will become public and objective knowledge, as long as it can be understood by everyone who is able to read the formal codes. Hence, rational processes that are delegated to machines are a form of an objective theory of action. Holling and Kempin[2] call them "implemented theories," whereby they define "implementation" as the general execution of a formal process by a machine. This makes implemented theories a subset of those theories that are characterized by being *operable*. Both authors refer to the fact that there is a long tradition behind underestimating the formal – but not only in the arts and humanities, which viewed formalism as a limited tool used in the natural sciences, but in mathematics as well, which also attaches no social importance to its codes. However, both sides overlook the real significance of the formal as a social integration apparatus, and as an implemented theory. Although formal codes spread like wildfire with the increase of computers, they are still invisible and unobserved.

We would like to examine how it is possible for thought processes to be conveyed to machines in the first place: the thought process first has to be translated into a mechanical process of action. This gives the formalization the decisive role of mediator. Formalisms serve the precise symbolic description of methods, whereby methods in terms of formalism are nothing more than rules for our actions that allow us to select a specific action from many different alternatives. Previously analyses are functionalized by formal structures and, thus, translated into a learnable inventory of knowledge and to a transmittable process. "Formalization is nothing more than the most manageable way of functionalizing that which has already been established; but it could also be a process of mechanization, because whatever can be formalized – meaning, what its application gains notwithstanding the insightfulness of the execution

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

– is already basically mechanized, even if the actual mechanisms needed for its storage and regulated association did not previously exist. All methodologies want to arrive at an unreflected repeatability, a growing archive of prerequisites that might always be in play, but do not always need to be updated.[3] We are now experiencing how our world is becoming ever more dominated by algorithms and – in the words of Frieder Nake – how everything, ultimately, is in fact succumbing to the power of algorithms. Yet in this context, the true significance of the code is its function as a social integration apparatus. That which has been coded, transmitted to thousand of machines, and has become a part of our day-to-day lives because it has been repeated again and again will stabilize and ultimately become a culturally unquestioned sediment. As an implemented and executable theory, the formal codes enter into an innovative and powerful relationship with the machine. Computers are not only the passive carriers of symbols, they are also active generators – symbols generate symbols. Yet it would be wrong to think that we would get the same phenomena back that we entered into the program during the design phase. The abstraction process needed to arrive at algorithms and operating symbols effectively runs backwards during the execution of the code, but not identically. If abstraction serves to withdraw, generalize, and purify the phenomena from insignificance and ambiguity, then, during execution, the program's corresponding interfaces recharge the results with the unintentional, the unfocused, and the ambiguous – for instance during the transition from image generating algorithms to the image itself. But that which has been charged is different to that which was discarded by means of abstraction, that is, on the path to algorithm.

Universality via Coding

"The importance of the universal machine is clear. We do not need to have an infinity of different machines doing different jobs. A single one will suffice. The engineering problem of producing various machines for various jobs is replaced by the office work of "programming" the universal machine to do these jobs."[4] *Alan Mathison Turing*

3 Translated from Hans Blumenberg, *Wirklichkeiten, in denen wir leben*, Ditzingen, 1981, p. 41.
4 Alan Mathison Turing, "Intelligent Machinery" (Prologue), in *Machine Intelligence 5*, ed. by Bernhard Meltzer and Donald Michie, Edinburgh, 1969, p. 7.

In the famous essay "On Computable Numbers, with an Application to the Entscheidungsproblem" Turing described the principle of the universal machine as follows: "It is possible to invent a single machine which can be used to compute any computable sequence. If this machine U is supplied with a tape on the beginning of which is written the D.N.[5] of some computing machine M then U will compute the same sequence as M."[6] Applied in a practical manner, this means that the list of instructions is coded for a specific Turing Machine *M* and writes this on the front of the tape. Coding means that, with a binary universal machine, we show the program as a sequence of zeros and ones, and not simply the data. This piece of tape is now the first part of the input for the universal Turing Machine *U*. It processes the remaining part of the tape by carrying out the instructions on the beginning of the tape and thus exactly imitates the specific machine *M*. In principle, today's programmable computers function along the same lines. They simulate the behavior of a specific machine by the program code being together with the data in the memory. Without software, the universal machine may be open for any conceivable behavior, but it is more or less incapable of action on its own. The universal machine begins behaving like a special machine, which is defined by the program, only after starting the program. Universal computers consist of two mechanical levels:

1. Hardware that is bound to conventional material.
2. The program codes that are only subjected to algorithmic laws.

The first level – the hardware – is bound to nature and obeys the laws of our physical world. It needs energy, is subjected to the aging process that will eventually lead to functional failure, and can be damaged by applied forces such as mechanical shocks, magnetism, and voltage surges. These material machines can execute logical elementary operations on the basis of physical principles. A second level sits on top of these electronic (mechanical, optical, and so on) machines that is not subject to physical laws, but instead is bound to human rationality and the limits of the formal. These code-based machines are beyond material weaknesses. They only know failures that can be traced back to a logical error on the part of the programmer. The code-based machine

5 D.N. = description number.
6 Alan Mathison Turing, "On Computable Numbers, with an Application to the Entscheidungsproblem," p. 241, quoted in: http://plms.oxfordjournals.org/cgi/reprint/s2-42/1/230 (accessed May 4, 2010).

brings the world of algorithms to fruition, it controls the performance of the bottom level, and is strictly bound to formal languages and the laws of human logic. New physical worlds, which are ruled by new and different natural laws, can be described on the machine's algorithmic level. We can delegate our reflections on the physical world to machines in the form of algorithms, and reencounter this logical construct in the simulation, but in completely new ways – no longer as a formally logical description, but as a total experience. The limits of this second machine go beyond our physical world; they lie in the limitations of codes and the human capacity to invent algorithmic worlds.

The Reflexive Properties of the Code

Both data and programs are coded in the programmable machine on the basis of identical symbols – consequently, these symbols are able to constantly enter into new relationships with one another. This is different in arithmetic, which makes a strict distinction between *operands* and *operators.* The computer can mechanically process data as well as programs. In architecture, for instance, architects design buildings, but in computer architecture, other computers actively help develop the next generation of computers. This is only possible because the code can switch functions. We can treat codes as data, but we can also actively manipulate them, meaning, they can become calculating units themselves. Any programs we download and start from websites are examples of *code as operand* and *code as operator.* The code is passive during the web transfer and treated as data by the network software. When the program is started on the computer, the code switches from operand to operator. As an operator, the code now manipulates data. The reflexive ability of the code is what enables the machine to become a practical entity. Without self-referential processing mechanisms, working on the computer would still be about zeros and ones. In order to explain the reflexive relationship between *code as operand* and *code as operator,* we would like to introduce two slightly complex examples: the compiler, and genetic programming.

Compilers are programs that contain symbols as entries (a program *a*) and use these to generate new symbols (program *a'*). The importance of the compiler's function is its ability to complete work in the first phase so that, in the second phase, the program can shift from the status of operand to operator. Compilers have to exist as a machine program in order to be run by a machine.

Creating them is a very time consuming and complex development process. Every compiler is specific to a single programming language *n* times a single machine language *m*. Therefore writing a compiler requires an amount of effort that increases exponentially with the numbers *n* and *m*. Clever use of the code's reflexive properties allows the compilers to translate themselves to a certain extent. The procedure is know as bootstrapping, and was developed very early in compiler history: from a compiler *a* designed for a specific implementation language and a specific Computer *A*, it easily creates a Compiler *b* for this implementation language on a Machine *B*.

1. In Compiler *a*, which already exists as code in the implementation language, the compiler creates the appropriate commands for Machine *B* everywhere the machine code for Machine *A* is created. This procedure produces the compiled code, named *a'*.
2. This code is translated by *A* and *a* into the Machine Code of *A* and results in a compiled code named *a"*.
3. *a"* is a program that can run on Computer *A*. Yet, this program receives *a'* as input and creates Machine Code *b*, which can run on Computer *B*. *B* is the desired compiler that runs on Computer *B* and creates programs that can also run on *b*.

The second example in this context is genetic programming. It belongs to the class of evolutionary algorithms that, as heuristic optimization algorithms, can offer good results even if there are no available closed solutions. They utilize the same properties of biological evolution based on populations of individuals that pass their genetic code on from generation to generation. The properties of each individual are encoded as genotypes. They represent the genetic construction of the individual and provide its blueprint. Generations are reproduced by a process of identical copying, the mutation of single chance positions of the code or the crossover of substructures between two individuals. Changes in the characteristics of an individual can only occur on the genotype (the chain of characters), but selection of the fittest according to the fitness function takes place on the phenotype. The phenotype is the physical manifestation of the genotype, that is, the specific characteristics of an individual in its environment. Special genotypes such as computer programs are worked on in genetic programming. The genotype in this context is the program code und the phenotype is the program's performance when it is running on a computer.

We can recognize even here the mechanism of the shift between *code as operator* and *code as operand.* The genotype functions as an operand for the basic genetic algorithm, and the phenotype of this code as the operator.

Automatic Self-Reproduction

This principle of genetic programming is also an example of the splitting of the universal computer into two machines, as described above. We acquire codes by means of evolutionary algorithms that are capable of manipulating themselves and further developing their performance. The lower level of the machine, the hardware that runs the genetic program, remains completely unaffected. In the following we will demonstrate that the spread of the code's self-manipulation strategies to the physical world is also used in the classic version by John von Neumann.

We first have to point out the sharp difference between self-replication and self-reproduction. Self-replication is a process by which an object creates a copy of itself. With regard to automatic replication by machines, this can be executed as a deterministic process in which it is particularly important that no flaws occur. Codes that exist as an explicit description of the machine are not needed in this case. In contrast, there is a possibility for diversity in "self-reproduction" as it implements a process of development that not only accepts, but also seeks variation and difference from generation to generation. Self-reproduction therefore is also a self-maintenance system that follows the Darwinian theory of evolution. Self-replication would not be sufficient to simulate life because it does not have any inherited mutations. To perform automatic self-reproduction, it would be useful if machines processed more than only the material in their environment to make copies of themselves, but also deal with specific information that exists in the form of codes. This does not mean any codes, but rather information with reflexive properties, that is, descriptions of their own construction. The first such process that allows more than simple self-replication, but also solves the evolutionary problem, can be traced to the mathematician John von Neumann.

This ability initially seems like a *circulus vitiosus,* because we would expect that the level of complexity of the systems that build other systems would decrease from *parents* to *offspring.* Automat *A* would need more than a complete description of Automat *B* in order to build *B*. It would also require different equipment that could interpret the description and carry out the construction work. Yet the initially plausible assumption that the complexity of self-building automats would decrease over generations is actually against the laws of nature's self-sustainability. Organisms reproduce and create new organisms that are at least as complicated as themselves. The level of complexity, as we know, can even increase over the course of a long evolutionary period.

But how can the general logical principles that allow automats to self-reproduce, and even increase in complexity, be described? Von Neumann's conclusion maintains that a minimal level of complexity is needed to allow automats to self-reproduce or to even create higher beings. Below this level, the automats are degenerative, meaning automats that build other automats are only capable of creating a less complex automat. John von Neumann observed five different models of self-reproduction. Arthur W. Burks named them the kinetic model, the cellular model, the excitation-threshold-fatigue model, the continuous model, and the probabilistic model.[7] The Turing Machine could not solve the problem of kinetic self-reproduction that we will examine more closely below, because it could only produce symbols in a piece of tape. However, with the kinetic model, von Neumann targets automats that build other automats, in other words, that do more than just manipulate symbols. They can also build hardware. The basic principle behind the kinetic system, von Neumann's focus of study since at least 1948, is not easily summarized. Von Neumann's self-reproducing automat is an aggregate consisting of three automats and their particular descriptions (blueprints).[8] First, a construction Automat *A* that is able to build any Automat *X* when given a description of Automat *F (X);* second, a copying Automat *B* that can make an identical copy from any description; and third, a control Automat *C* that monitors the interaction between Automats *A* and *B*. To begin, it activates Automat *A,* which immediately starts building Automat *X* according to the description *F (X).* Then, it activates Automat *B,* which produces a copy of *F(X).* This copy introduces C into the new Automat *X,* which was just built by A. Finally *C* separates the newly built Construction *X + F(X)* from the construction automat.

7 See John von Neumann, *Theory of Self-Reproducing Automata,* ed. and completed by Arthur W. Burks, Urbana, 1966.

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

We can now call the entire Aggregate $A + B + C$ with D and get the desired self-reproduction Automat $D + F(D)$, whereby $F(D)$ is again a description of Automat D. What is significant about the von Neumann principle of self-reproduction is that it is impossible without coding. The machines have to be diverted by the coding of their own construction in order to be reproducible. Yet this means that Automat $D + F(D)$ does not only replicate itself – it actually possesses other important properties. We see for example that coincidental changes in $F(D)$ refer to characteristics that appear in biology in relation to mutations. Most random changes to $F(D)$ will render the automat inoperable. If there are enough changes, there will undoubtedly be several among them that will lead to new functional automats. These new automats possess different characteristics than their predecessors.

No changes in the description of the automat allow for a construction process that reproduces automats as well as by-products. Let E be the automat with the corresponding description $F(E)$; if we feed the above self-reproducing Automats $D = (A + B + C)$ with $F(A+ B + C + E)$, a new Automat $(A + B + C + D) + F(A + B + C + E)$ is created. The new automat does not only produce a copy of itself; it also generates an Automat E.

This play of automats no longer requires a user. Their duplication and further development becomes self-perpetual the moment the process is started. Ignoring for a moment the dangers inherent in surrendering all human control, this form of automatic autonomy is nevertheless undesirable for most work processes. Tools that expand the user's possibilities, and not the machine's, are far more interesting. The important issue here is how much will a programmer's decisions and the deterministic operating code be able to influence the desired result. How much creative space will software tools allow their users?

The Openness of Codes

A process can be formally described when it fulfills the conditions of writing, schematizing, and freedom of interpretation.[9] Writing means that the process can be expressed using symbols; schematizing means that the procedure can be described by a set schema and general processes (also algorithmic); and freedom of interpretation means that we are completely open to choose symbols

9 See Sybille Krämer, *Symbolische Maschinen*, Darmstadt, 1988, pp. 1–4.

and designations since we no longer refer to what the symbols represent while doing the operations. The process of formalizing has removed our need to understand on the level of symbols. A layperson can only understand what the symbols mean if he or she knows the key to the code. All of the three above-listed conditions of formalization are automatically fulfilled if the operation can be expressed in a program language readable by the machine. Formal structures, according to the present notions, leave little room to the imagination. Formalizing means that data is recorded as exactly as possible and that any misinterpretations or ambiguities can be rectified. The interesting question is where there might be space for participation and imagination when working on or with the computer. Does the formal, basic pattern of the computer and its programming only lead to stiff and uncreative results? Are architects who work with programs such as these at all free to decide and, hence, design, and if yes, to what extent? Have not all decisions once made by architects been specified in the tool, long before any individual production takes place? Is there still any space that allows for free designs that go beyond the settings made by the programmer?

When we speak of a program code being open, we are speaking very generally about features that allow new space for the users' action and interpretation, and that create a certain amount of flexibility and permeability for their intentions while working with the program. The program text, however, will always be the explicit and unambiguous given. On this level of description, freedom can only be achieved by introducing the self-changing strategies discussed above, like learning methods, self-references, or the general reflexive properties of the code. New things, or that, which the programmer assesses as unexpected, can only develop on the performative level of the code when the code writes itself. In fact, the openness of an application can be very easily achieved without any explicit changes made to the code in conventional applications. To achieve openness – according to the theory represented here – the qualities that should remain open should not be formalized. Here is an example taken from architecture.

Umberto Eco classifies architectural codes into syntactic and semantic categories.[10] Syntactic codes emulate the knowledge of constructions. They

10 Umberto Eco, *La struttura assente. La ricerca semiotica e il metodo strutturale*, Milan, 1968. The subsequent quote is translated from the German edition *Einführung in die Semiotik*, Munich, 1994, p. 329.

include beams, ceilings, arches, bearings, columns, and prefabricated concrete supporting structures. However, these syntactical codes do not yet refer to a function or a denoted space. This is achieved by semantic codes, which include primary functions such as roof, terrace, stairs, and window, but also domestic ideologies (communal room, day and evening zones, dining room, waiting room), *social typological genres* (hospital, villa, school, castle, train station), and *spatial typological genres* (round or cross shaped floor plans, labyrinths).

An architecture that works with codes such as these, according to Eco, cannot offer anything to their users that would not catch them off guard. "The common viewpoint regarding all of these codifications is that they give form to previous solutions, which means they are codifications of message types. [...] Hence, the codes [...] would be nothing more than iconic, stylistic, or rhetorical encyclopedic dictionaries. They do not offer any generative possibilities but rather a complete diagram, not open forms that can be discussed but hardened forms and general relationships of an unexpected kind. [...] It is not true that a few empty and purely differential forms of architectural import (columns and beams) make just any architectural statement: they specifically make the architectural statement to which Western civilization has accustomed us according to certain statistical and dynamic criteria and certain rules of Euclidian geometry. Even if they are more stable and more resistant to wear than other control systems, they still force us to operate within a specific architectural grammar. At least it is possible to find them codified under the term structural theory."[11] Architecture is not the only field that faces the challenge described by Eco, but anyone developing tools for open design processes. The implemented syntactic and semantic codes – called ontologies in the field of computer science – already decide which results are to be expected. If one wants a surprising result in relation to certain concepts, theories, and aesthetics, then they should precisely not be entered into the formalism of his or her system. If one does not want a building constructed of beams, ceilings, columns, and walls, then these should not be used as the basic elements of this design system either. What seems trivial here is nevertheless little reflected or even overcome in the available tools. Forfeiting familiar categories and concepts is certainly not very difficult; the true challenge is shifting them to a different level of abstraction. Categories that should remain open need to be replaced by more abstract concepts, so they can be reconstructed as unique cases – they will remain unformalized, but are able to be created and reflected in the work process.

11 Ibid., p. 329ff.