CHAPTER 5

What Is Software?

A cultural definition

The previous chapters have spoken of "computations," meaning calculation and algorithms performed either in the medium of language itself or with the help of mechanical or electronic devices. For many speculative computations, like those in the *Sefer Yetzirah* with its obscure mechanical device, it's difficult, if not impossible, to draw a clear line between symbolic-imaginary and material processing, software and hardware.

What is software? Simply defined as algorithms, software would fail, first of all, to encompass the vast speculative imagination from magic to codework in which computational code rarely is pure algorithms, but a metaphorical hybrid. A strictly formal definition of software would also fail to describe the line of the Steven Seagal movie, "300,000 pages of code. Or 60 minutes of triple-X rubber-and-leather interactive bondage porno." That line shows that such a cultural understanding isn't far-fetched, but is already street wisdom. Why, after all, did the mathematician John W. Tukey invent the term "software" in 1957 given that the term algorithm, phonetically derived from the 9th century Persian mathematician Muhammad ibn Musa al-Khwarizmi, existed centuries before? Obviously, software referred to algorithms as control logic that was abstracted from the machine. The Unix operating system which runs on almost any kind of hardware[1] is a prime example of software as an abstraction from hardware. But just as the cultural history of computation is rich with metaphorizations and meanings inscribed into formal processes, the same is true for software. In 1970, only 13 years after Tukey's coinage, Jack Burnham's *Software* exhibition appropriated the term metaphorically. In a 1968 essay *Systems Esthetics*, Burnham observed that a written piece

[1]The free Unix clone NetBSD supports runs almost any existing hardware platform including IBM-compatible PCs, palmtops, video game consoles, old Amiga and Atari home computers and proprietary Unix servers.

of conceptual artist Donald Judd "resembles what a computer programmer would call an entity's /list structure/."[2] Reading these semblances, Burnham did not only adopt software as a metaphor for conceptual art, but also turned, and aestheticized, computer software into concept art.

Software as practice

Just as, for example, literature is not only what is written, but all cultural practices it involves—such as oral narration and tradition, poetic performance, cultural politics—software is both material and practice. As the verb "to google" for using the Google search engine shows, or in their computational sense, "to browse," "to chat" and "to download," human practices are born out of the use of software. Googling is nothing but the shorthand for using the web-based client-server software written by Google corporation's programmers. In this sense, software is no longer just machine algorithms, but something that includes the interaction, or, cultural appropriation through users. This appropriation is more than just a cybernetic human-machine interaction and what computer science and media theory often reduce to pointing, clicking and other Pavlovian responses within the restraints of a programmed system.—The same reductive understanding of interaction has turned "interactive art" in its common phenomonen of behavioral video installations into an artistic dead-end.—True interaction with technical systems involves creative use and abuse outside the box, metaphorization, writing and rewriting, configuring, disconfiguring, erasing. All these practices also make up software.

It wasn't just artistic appropriations that inscribed metaphors into software. High-level, machine-independent programming languages and operating systems such as C and Unix gave birth, around the same time, to a culture that gradually detached software from the concept of code running on a machine. Through program code listings in books and computer magazines, source code snippets and patches exchanged in electronic networks or even oral conversations, software took up a life of its own. The results were political-philosophical movements like Free Software, programming puns such as recursive acronyms, hacker slang that mixed English and computer language constructs and poetry in computer languages such as Larry

[2] Jack Burnham. Systems Esthetics. *Artforum*, 9 1968. http://www.arts.ucsb.edu/faculty/jevbratt/classes_previous/fall_03/arts_22/Burnham_Systems_Esthetics.html. [**16**]

Wall's first Perl poem from 1990. Free software—in the GNU understanding of an embedded value that is not only engineering freedom, but ontological freedom—is perhaps the strongest example of a cultural and philosophical notion of software. An artistic understanding of software also abounds in computer science from Donald Knuth's *Art of Computer Programming* to Paul Graham's recent *Hackers and Painters*,[3] although it might be based on a narrow understanding of art as high craftsmanship. To no longer define software as just algorithms running on hardware helps to avoid common misunderstandings of software art as some kind of of genius programmer art. If software is a broad cultural practice, then software art can be made by almost any artist.

Software versus hardware

Aside from the blurriness of software as a machine process and software as a human cultural practice, the technical distinction between software and hardware is blurry itself. Is instruction code hardware once it is burned into an EPROM, is it software when it is stored in an erasable flash ROM? What about microcode, computer programs stored right within chips in any modern CPU? Or chips like the Transmeta Crusoe which has only minimal hardware wiring and implements its CPU instruction set—like Intel-compatible x86—solely through an embedded emulation software (written originally by Linux creator Linus Torvalds)? What about the BIOS or firmware of computer mainboards, graphics cards, network adaptors without which this hardware simply isn't operational? Isn't it a totally arbitrary distinction whether a circuit is hardwired into the layout of chip transistors, or whether the same logic is stored within a memory chip? The definition of hardware, in turn, is not less doubtful. The first modern computing hardware, the Turing machine, did not materially exist, but was theoretical and imaginary. The same applies to Donald Knuth's *Mix* computer. The cultural history of computation proves that hardware can be metaphorical when algorithms run on any imagined material including the entire cosmos in Quirinus Kuhlmann's speculation. Still, in the end the distinction between software and hardware relies on Cartesian categories: Is, for example, a human brain that performs a computation a piece of hardware?

[3] Donald E. Knuth. *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, 1973-1998. [**55**], Paul Graham. *Hackers and Painters*. O'Reilly, Sebastopol, 2004. [**40**]

If the duality of software and hardware needs to be suspended, it follows that the notion of software as immaterial versus hardware as material must be suspended, too. The difference between materiality and immateriality exists within software itself: an algorithm is material in its stored, coded form and most of its cultural practices. The immaterial component might be the imagination and phantasms invested into the software, that inserting a CD-ROM, to refer to the previous example, might blow up the earth or get you a sexual turn-on (or turn-off perhaps). Some software chiefly or purely exists in imaginary form:

- vaporware, the constantly promised "stable upgrade" of a crashing computer program for example, or the ever-procrastinated new version of a piece of software, like the computer game *Duke Nukem Forever*;
- hoax viruses, i.e. E-Mail "memes" which instruct gullible readers to erase critical system files on their computer;
- imaginary virus infection; hardware failures or human mistakes wrongly interpreted as computer virus infections.

If the location or even existence of some software isn't quite clear, if a piece software isn't code running on machine because it appears as pseudo code in a book or is, like most Perl poetry for example, not even a working algorithm, then a technical definition of software is too limited. In the end, "software" and "computation" can't be strictly differentiated from each other. The cultural history of software is the cultural history of computation.

Conclusion

Software, it follows, is a cultural practice made up of (a) algorithms, (b) possibly, but not necessarily in conjunction with imaginary or actual machines, (c) human interaction in a broad sense of any cultural appropriation and use, and (d) speculative imagination. Software history can thus be told as intellectual history, as opposed to media theories which consider cultural imagination a secondary product of material technology. This booklet took language computations as its primary examples mostly because language can be computational in itself. Thanks to its abstraction and grammatical structure, it also expresses computation better than any other symbolic form. Programming languages, with their modified English, are the proof. But or architecture, too, could serve as the main examples in a cultural history of computation since they both combine formal instruction with imagination.

The cultural history of computation shows that it is as rich and contradictory as that of any other symbolic form. It encompasses opposites, algorithms as a tool versus algorithms as a material of aesthetic play and speculation, computation as inner workings of nature (as in Pythagorean thought) or God (as in Kabbalah and magic) versus computation as culture and a medium of cultural reflection (starting with Oulipo and hacker cultures in the 1960s), computation as a means of abolishing semantics (Bense) versus computation as a means to structure and generate semantics (as in Lullism and Artificial Intelligence), computation as a means of generating totality (Quirinus Kuhlmann) versus computation as a means of taking things apart (Tzara, cut-ups), software as ontological freedom (GNU) versus software as ontological enslavement (Netochka Nezvanova), ecstatic computation (Kuhlmann, Kabbala, Burroughs) versus rationalist computation (from Leibniz to Turing) versus pataphysical computation as the parody of both rationalist and irrationalist computation (Oulipo and generative psychogeography), algorithm as expansion (Lullism, generative art) versus algorithm as constraint (Oulipo, net.art), code as chaotic imagination (Jodi, codeworks) versus code as structured description of chaos (Tzara, John Cage).

Computation and its imaginary are rich with contradictions, and loaded with metaphysical and ontological speculation. Underneath those contradictions and speculations lies an obsession with code that executes, the phantasm that words become flesh. It remains a phantasm, because again and again, the execution fails to match the boundless speculative expectations invested into it. Cultural and political semantics result merely from its dull formalisms and their interference with daily life, from account balance statements to "end-user software." Formalisms create semantics in a wholly different way than people expect from an allegedly "intelligent machine." Computers therefore exist, as hacker wisdom says, to solve problems which we would not even be aware of having if not for the computers themselves.