Dietmar Eberle

9×9

A Method of Design

From City to House Continued

Birkhäuser
Basel

MANAGING EDITORS

Florian Aicher
Dietmar Eberle

EDITORIAL TEAM

Franziska Hauser
Pascal Hofmann
Marcello Nasso
Stefan Roggo
Mathias Stritt

Architecture and Computer Science, Computer Science and Architecture

Michele Lanza
Marcello Nasso

M.N.| This book is an attempt to systematize the teaching of architecture. Are there parallels to the methods and topics seen in the computer sciences?

M.L.| In software engineering, as in architecture, we deal with the question of how to build, maintain, and evolve large systems.

Unlike in architecture, software has no place–no up and down, no left and right. Everything is everywhere and nowhere at once. There are no distances and the place where something is done or written is not relevant. At run time, when the system is executed, all distances collapse into practically nothing.

Yet there is indeed a structure. The entire thing must be organized and given a structure. This is why we speak of *software architecture.*

M.N.| In computer science, the structure is defined by the software architecture. What does software architecture involve?

M.L.| It involves the structuring of systems: How systems are divided into subsystems and again into sub-subsystems and how these systems and subsystems communicate with each other. That is software architecture. It creates an order.

M.N.| This is what architecture does as well. A city or a building is conceived in the same way–a city by using master plans and building types, buildings by using construction systems and building components.

What differentiates software engineering from architecture besides its lack of place?

M.L.| The other difference I see is that in architecture there is always a thought process that leads to a concrete result in the end. For the engineer and the master builder who build the system, these are the plans. This means that in the end there is a model and an implementation of the model. In software engineering, this difference doesn't exist.

Nobody designs a system that is implemented by someone else. Although this is attempted, what is actually happening is the implementation of a model. The source code is what is physically and effectively written in the end. In this sense, it is a model of the system.

M.N.| What form does this model of a system have?

M.L.| Currently, it is in writing.

M.N.| Does this mean language?

M.L.| Exactly. What is presented at the end, what the programmer sees, is many lines of text, written in a specific programming language, which the machine can understand. And therein lies the intellectual complexity of programming. What programmers actually do is write structured systems in a complex manner.

What they actually see are many lines of source code, packed in files.

The problem is that millions of lines in thousands of files are needed in order to master the necessary complexity. Abstraction and the mastery thereof are the skills that, in the end, make a truly great programmer. Programming is abstraction at a very high level.

In his book *The Mythical Man-Month,* Fred Brooks says that programmers, like poets, are not far from pure thought. They build castles in the sky that are made of air, simply by allowing their imagination to prevail. He wrote this in 1975.

M.N. | That is a very personal vision of the big picture. In order to build a large system, a collective is necessary: many different actors with shared visions and the goal of together creating a large whole. Seen from this point of view, the metaphor of the romantic poet no longer seems to work.

M.L. | Two thirds—and thus the great majority—of complex software projects involving multiple actors go awry. They don't mess up for technical reasons—they mess up because of human error.

M.N. | Because the actors can't make themselves understood?

M.L. | Yes, due to communication difficulties—from person to person, not from person to machine. The topics that must be communicated are simply too abstract.

And then, human nature must be factored in as well. Something we call Conway's Law says that systems designed by organizations are constrained to produce designs that are similar to the communication structures of the organizations that write them. If you are in, you understand; if you are out, you don't.

There is a system that organizes things—which parts go where. That is software architecture.

Another problem is the unbelievably high turnover that takes place in the software industry. Programmers change their jobs with great frequency and leave artifacts behind that other people, people who did not write these things themselves, then have to understand. It's as if the architect of a building would change five times while it's being built.

M.N. | The question is, can "good" systems overcome this, and is there is a super-system in normal institutions?

M.L. | Usually not.

M.N. | But there should be one—an overarching structure that acts as a net to catch knowledge.

M.L. | That only exists in organizations that can afford it. The key term here is *mission critical software.* NASA, for example, functions this way. Launching a rocket costs billions. If you are doing it, you have enough money to nail down a near perfect system. But it requires more time and large sums of money.

In the modern software industry, where everything is defined with marketability in mind, it's not affordable. It's not an advantage to write good or nice software, or to write beautiful software. The mere fact that the software is being written is already highly meaningful.

The first software engineering conference was held in 1968 in Garmisch, sponsored by NATO. There, the term *software crisis* was brought to the forefront. This was in recognition of the fact that the software industry was in a crisis, in a crisis of complexity. This crisis has not ended. It is still there and has grown even stronger. It's a vicious circle: The better the means of writing software become, the longer the systems live.

M.N. | Not how something is done is interesting, but that something is done. It is not about the inherent beauty of a system, but that the system has even been built. Unfortunately, this is familiar in architecture as well.

And yet: What is beauty anyway?

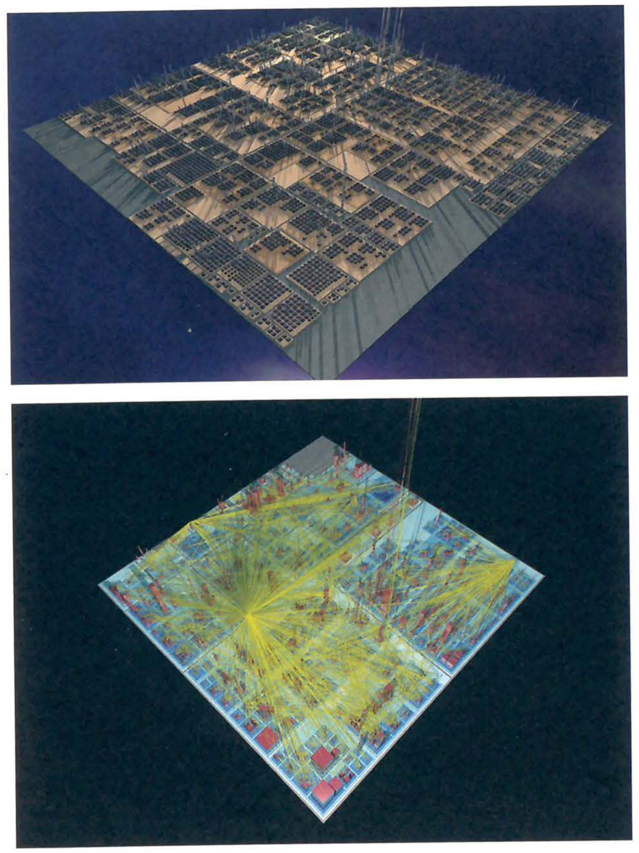
M.L. | For me, designing software means keeping all that is essential and throwing out all else.

Simplicity is beautiful. Simple code is beautiful. In Richard Gabriel's book, *Patterns of Software,* he talks about habitability in a similar context—the habitability of software. In this book, he says that there are systems in which it is wonderful to program, that are beautiful to look at, and which are comfortable to be in.

It's about how easy it is to make changes. Changes constantly have to be made. Ugly systems are systems in which changes, no matter how small, waste a huge amount of energy. In a beautiful system, it is clear what one must do, when one must do it, and how to do it. There is beauty on a linguistic level. There are beautiful lines of code where someone use the right variables and named the things as they should be named. It is also possible to write ugly things, by using variables that are really something else, or simply by giving things the wrong names. This makes systems much more difficult to understand.

In extreme programming, there is also the concept of egoless programming. This means that one doesn't write for oneself, but for someone else. One therefore tries to write in a way that is as easy as possible for someone else to understand.

M.N.| But as an ordinary user, I never see what the programmer has written. The result is communicated to me through an interface and thus in an entirely different way.


Michele Lanza, visualizations of the codes and metrics of different dimensions using various languages (Java, C++, Smalltalk, CSharp). The visualization takes the form of a city, improving general comprehension and potentially enabling any problems that arise to be more quickly understood. Each subsystem is a city district, within which buildings represent each individual file in the system.

M.L.| Exactly, now we are talking about the skin. This can be reduced to what we call the user interface. This is the part of the system seen in the end by the user. What I mean is that even if, for example Windows Vista is 80 million lines of source code, what the user sees at the end is windows, buttons, and graphic design—it's just the skin of the system.

M.N.| Isn't using a user interface to operate a system relatively new?

M.L.| Yes. The very first computers were not user interface-based. The first user interfaces showed up in the 1960s and even more in the 1970s. This is due to the fact that, before then, there were no users. The few people who used computers back then had the word computer in their job title. They knew how to use them and thus didn't need a user interface. It was not until computer science became mainstream in the 1980s and 1990s that user interfaces really became necessary—for people who wanted nothing to do with programming: the users.

M.N.| The interface is comparable to what we call the skin or the envelope in architecture. Another analogy is theater—in theater, it would be the stage, where everything happens.

M.L.| I understand what you are getting at. Your allegory describes the perfect interface. It would be what you don't see. It would be ideal to use an everyday language, without resorting to the crutch of an interface. But the machine just isn't smart enough for that. In order to obtain a such understanding, the machine would have to be intelligent—but it isn't.

M.N.| What is intelligence?

M.L.| Self-awareness. It has no self-awareness. The machine is not aware of itself.

M.N. | HAL 9000, the fictitious computer on the Spaceship Discovery in Stanley Kubrick's film *2001: An Odyssey in Space* from 1968, has a consciousness. The film is built up so that in the beginning, the machine is only an abstract technical construction, but then through the course of the film it develops an increasingly clear sense of self-awareness.

M.L. | At this point in time, that is simply impossible. There are people who believe in what we call singularity, which refers to machines developing a sense of self-awareness. There are people who say that this could happen within the next 15 to 20 years. But there were also people who said the same thing in 1965, and more than 15 years have passed since then. In my opinion, it currently still doesn't look possible.

M.N. | And so much for insight on the state of computer science. The parallels to architectural development are obvious, however, differences between the two disciplines are also becoming clear. Let's focus on these.

M.L. | Another difference to architecture is that there are no borders, no physical borders. There are thus no recognized rules that determine how software systems must be structured. There is no rulebook that one can or should follow. There's no difference between good and evil, so to speak. In the end, what the user sees, is in any case something totally different. One can design a system incredibly badly; but if the user interface is convincing, then the system is convincing.

M.N. | I understand: There are no rules and regulations. Are there schools of thought, methods, or theories?

M.L. | There are process theories. These relate to how one can methodically approach the design of a system. In these fields, there are also models. There is, for example, the waterfall model. This is when one thing is done after the other.

Waterfall is in reference to the methodology. Waterfall. Software engineering is often compared to conventional engineering or to civil engineering. In this, the metaphor of building a bridge is used. It is necessary to coordinate numerous persons. In the end, a result is produced that is useful to humans in some way.

But the difference that causes the metaphor to fall apart is this: When you start to build a system, the context then changes within a very short period of time. This happens continually. This means that when we describe designing a software system with the metaphor of building a bridge, we have to add the following: While the bridge is being constructed, the landmasses the bridge is meant to join are continually shifting. If you rigidly stick to a plan, you risk the end result being a bridge that joins nothing.

The metaphor only works if it is clear from the very beginning that the context will not change and when all requirements are clear from the very beginning. However, in the last fifty years of the field, it has become evident that it is impossible to fully anticipate the context. One has noticed: The context is continuously changing.

However, a different approach is also possible, one that emerged in the 1990s. It is called agile software development and encompasses various methodologies, for example, extreme programming. This recognizes that change is the most important factor and that one should not try to build systems that are resistant to change. Change is accepted as matter of fact.

Since this, various means of building systems have existed. Most of them follow an onion model – one creates skin after skin and in this way creates interactive steps or types that are strongly based on prototypes.

How is this different from architecture? In computer science, prototyping is cheap, because it's not physical. One can take a prototype and continue to develop it. In architecture, this would mean that one would start constructing a building in order to understand how it should not be done, and then tear it down again in order to build it the right way.

M.N.| In architecture, models and systems are not the same thing. It is possible to develop prototypes into context-specific building typologies using models. The two and three-dimensional graphical representations and the building of physical models at various scales are the means of simulation we use to avoid as many mistakes and misunderstandings as possible.

In addition to this, a great number of things are made differently, depending on the site or craftsmanship, or choice of industrial production method.

A house in a village should be similar to the neighboring house. Each house has its own identity, yet it is related to the house next to it and is part of a greater whole. The same is true for the parts that make up the different buildings.

M.L.| That is different from in the computer sciences. To quote Fred Brooks again: "There are no two identical parts in a software system."

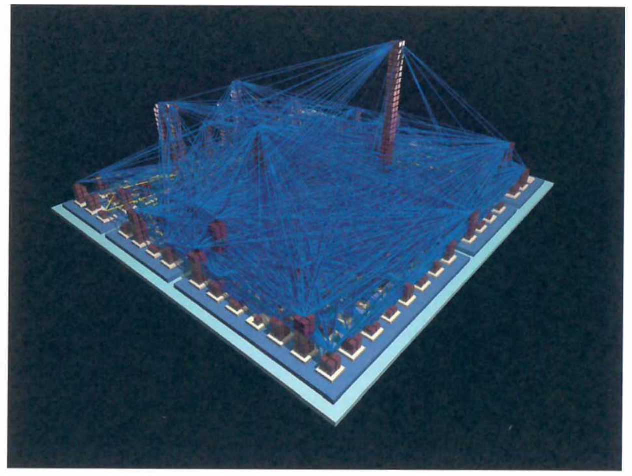
In architecture some parts are indeed identical. As you just described, buildings are based upon conceptually identical components. In computer science and software engineering, each line is written by hand.

M.N.| Are there any machines or programs that can support programmers by writing entire passages?

M.L.| Yes, there are. This is called model-driven engineering. One tries to create models and then a machine generates the source code at the push of a button. But the whole thing is just a vision that doesn't really work for two reasons: The first reason is that what it is possible to generate using such machines is by definition incomplete. It always has to be retroactively improved by hand. One can create skeleton programs, but the missing parts have to be inserted by hand. The other reason is the modeling languages. These are very complex—so complex that the models are more complex than that which the models are trying to model.

M.N.| What is the difference between a model and a program in computer science?

M.L.| A program is made up of source code. Source code is a high-level representation, a representation of what the computer is churning through on a deeper level in order to be able to execute something. A model is a mental representation of a program. It is not the program itself. Yet some say that the program is the model. And others say that the model can be used to generate a program.


Michele Lanza, a program visualized using CodeCity, a visualization system developed by a research team led by Lanza. Problems that could lead to a program crash are seen as very high buildings, hugely out of proportion in comparison to the other buildings. These "health maps" indicate the areas in which program architecture must be improved.

M.N. | For architecture, two analogous approaches can be discerned. There is functionalism on the one hand, which has been tied to modernism since the late 19th century. The arguments of this school are: The current use of the building, which in architecture we also call the program, generates the form. Rationalism argues against this, saying that the program is the result of the form and thus the structure.

Buildings with a set destination are often occupied differently all of a sudden. I'm thinking of basilicas—a building type developed by Romans. It was originally used for large public meetings, legal trials in particular. Then it became a house of God for Christians, since this type of structure permitted a great number of people to collect under one roof. The structural capacity defined the building typology, not its function.

M.L. | This is not a problem in computer science, since it isn't physical—the costs of building something are not comparable in software. And there is theoretically nothing lost.

From the 1920s to 1960s, computer hardware was much more important that the software. Nixon purportedly once asked how much the software that flew Apollo 11 to the moon weighed. Since this time, hardware has made increasingly significant leaps, and has become less and less important physically. In comparison, software is much more important, in fact endlessly more important. One has practically limitless space, an endless space within which to develop systems.

M.N. | We have talked about the absence of a place, the meaning of structures and systems, the necessity of the skin as an interface, and the ambivalence of models of programs. What about materiality?

M.L. | In this area, the crucial difference is plain and simply that, as mentioned before, there is no attrition. Materials are used: If people walk across a floor for hundreds of years, the floor is changed; a line of code, however, doesn't age in the same way, it never gets old. Architecture is subject to gravity, software isn't.

The system disintegrates nevertheless. This isn't because of the system, but because the reality outside of the system is changing. A system that is by definition immortal and interminable suffers from the fact that reality is not in a standstill. A system that is as it is and stays that way will very quickly become obsolete. It will no longer meet the needs of reality.

It must be adapted. There is indeed erosion, but it comes from without, not within. The elements are intrinsically nondeformable. With time, the system itself makes itself obsolete.

M.N. | What is the oldest system that exists?

M.L. | What is a system? When I think about modern software, there are no systems older than 50 years old that have survived.

Large banking systems are sometimes 30 to 40 years old. In computers, that is an eternity, dozens of programmer generations.

Language is added into all this. Software systems are written in a programming language. These are languages that were developed in order to be able to more easily communicate with the machine. These languages are continually being invented, and many simply die off and are gone. It is currently estimated that about 9,000 programming languages are in existence. Of these, 20 or 30 are actually used—the others are meaningless. New programming languages will rise up and replace the ones that are common today. But the systems will still be there. This is also a human resources problem: These systems, some of which are 30 or 40 years old, can only be understood by a very few people.

Another difference to architecture, to its materiality, is connected to the aspect of accountability. In modern software engineering, there is no accountability for a false application. In architecture, someone is held responsible for damages. If a software system has an error and, for example, an airplane crashes because of it, a programmer is still not held responsible for it. This has never been regulated.

M.N.| Is that a good thing?

M.L.| That's a good question. I think it's generally a bad thing. The field of computer science has failed as a whole when it comes to creating a professional image. Today, anybody can claim to be a computer person. I don't think anybody can just claim to be an architect and legally start building.

M.N.| Wouldn't it be enough to differentiate people with a high level of know-how from those without?

M.L.| When a field has no boundaries that define what falls within it and what is out of bounds, then it is difficult to continue developing it. If today, for example, there is a call for environmentally sustainable development, then it is clear to whom one can go in the field of construction.

When attempts are made to make the field of computer science more professional, nobody knows who to turn to. Only a very few are actually professionally educated. I think that the majority of source code on this planet was written by amateurs.

M.N.| It's no different in architecture. Only a very few architectures and cities were built by architects.

M.L.| There are a great many parallels between architecture and software engineering. They are both about complex structures. Complex structures that change over time, that must change, must be converted. These are structures that, at the end of the day, were built for people. I believe that architecture has adopted very little of use from software engineering while, in contrast, software engineering has taken on many useful aspects of architecture. I think the entire field still lies fallow●

Library of Congress
Control Number: 2018937456

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.dnb.de.

ISBN 978-3-0356-0632-4
(Hardcover)

ISBN 978-3-0356-0633-1
(Softcover)

This publication is also available as an eBook
(ISBN PDF 978-3-0356-1099-4)

and in a German language edition
(ISBN Hardcover 978-3-0356-0621-8,
ISBN Softcover 978-3-0356-0622-5,
eBook 978-3-0356-0662-1).

9 8 7 6 5 4 3 2 1
www.birkhauser.com

EDITORS
Dietmar Eberle
Florian Aicher

EDITORIAL TEAM
Franziska Hauser
Pascal Hofmann
Marcello Nasso
Stefan Roggo
Mathias Stritt
ETH Zurich
CH-Zurich

ACQUISITIONS EDITOR
David Marold
Birkhäuser Verlag
A-Vienna

PROJECT AND PRODUCTION EDITOR
Angelika Heller
Birkhäuser Verlag
A-Vienna

TRANSLATION FROM GERMAN INTO ENGLISH
Word Up!

PHOTO EDITOR
Dominique Jahn
CH-Zurich

DESIGN
Gottschalk+Ash Int'l
CH-Zurich

PRINTING
Holzhausen Druck GmbH
A-Wolkersdorf