Software Engineering

Ernst Denert

Abstract "A Passion for Software-Engineering." This was the headline of a 2015 newspaper article about Ernst Denert. And they were absolutely right. Ernst Denert is really passionate about developing software with excellent quality in a predictable and systematic style. Furthermore, he is very much interested in encouraging young people to study computer science or at least to learn how programming and digitalization works, as well as computer science students to focus on software engineering principles and software development. This chapter is a personal view of Ernst Denert on the software engineering discipline.

A personal view on the Software Engineering discipline.

1 1968

"What we need, is software engineering," said F.L. Bauer in 1968, and he organized the conference in Garmisch that founded our profession. The following is a translation of an excerpt from his report (available only in German) on this conference in the Informatik-Spektrum[1] 25 years later:

> One day, annoyed by the fact that in the end, nothing more than yet another pure academic project could emerge, I commented: "The whole trouble comes from the fact that there is so much tinkering with software. It is not made in a clean fabrication process, which it should be." Noticing that this shocked some of my colleagues, I added: "What we need, is software engineering." That made an impact.

[1]Quoted from F.L. Bauer: Software Engineering—wie es begann, (Software Engineering—How it Began) Informatik-Spektrum, Oct. 1993, page 259.

E. Denert (✉)
Grünwald, Germany

M. Felderer et al. (eds.), *Ernst Denert Award for Software Engineering 2019*,
https://doi.org/10.1007/978-3-030-58617-1_2

> The official conference report[2] recorded the following:
> The phrase 'software engineering' was deliberately chosen as being provocative, in implying the need for software manufacture to be based on the types of theoretical foundations and practical disciplines, that are traditional in the established branches of engineering.

The aim was that software development should become an engineering discipline with a systematic method of working, roughly according to the schema

plan—design—produce—check—install—maintain.

Just like a house needs a construction plan and a machine needs a design drawing, so a software system needs an architecture.

2 Software Architecture

Software architecture is the supreme discipline of software engineering. It covers the design of software systems as a whole, with their technical functionality and the interfaces to the outside world, as well as the internal structure at every level of detail. At a high level, software architecture is concerned with the construction of a system made up of modules, the interaction between these modules in executable processes, and the distribution of the system over hardware components. At a detailed level, software architecture is concerned with the design of individual modules: on the one hand, with their external view, that is, their interface to other modules, and on the other hand, with their internal structure, that is, which functions operate on which data.

In IT, the word "architecture" has become established as a "superior" designation for various types of structures, and this is also true in the case of software. Software architecture is a buzzword that is much overused. It has many interpretations, with the understanding sometimes being that architecture in itself is something good. That is not true—some architecture is bad. Sometimes we hear, "Our software has no architecture." That is also not correct; every system has an architecture. What is generally meant is that the structure of the system is poor or inconsistent, it exists undocumented only in the minds of some programmers or is completely unknown. Nevertheless, it does exist.

Such structures are frequently represented in graphic form with boxes and lines between the boxes, mostly without defining precisely what they mean. The boxes represent components, modules, functions, files, and processes; the lines and arrows represent various types of relationship, references, data flows, calls, and much more. There is often no consensus over them, not even within a project, a department, and

[2]SOFTWARE ENGINEERING, Report on a conference sponsored by the NATO SCIENCE COMMITTEE, Garmisch, Germany, Oct. 7–11, 1968, chapter 1, page 13.

certainly not within whole commercial businesses, and just as little in scientific institutions.

My definition of *software architecture* is that it is the static and dynamic structures that shape a software system, considered from three perspectives, referred to as *views*: the application, design, and program views. We think about the system differently in each view. And thinking needs language. All people think in their native language, musicians think in notes, architects think in drawings, medical professionals think in Latin terms, mathematicians think in formulas. And programmers think in programming languages. However, that is not enough to design software architecture; each view has its own terminology, concepts, and forms of presentation:

2.1 Application View

For this view, we have to understand the language of the users and think in concepts and the terminology of the application, regardless of the type of the application (business, technical, scientific, media, etc.). In *business* systems, this view is concerned with data, technical functionality, and business processes, as well as interaction with users (dialogs) and neighboring systems; in *technical* systems, the view is concerned with the physical machinery and control processes. Technical content is described with natural language and more or less formal representations.

2.2 Design View

This view is concerned with the modular design of a software system—that is, its construction from modules with their static and dynamic relationships. Graphical representations are very popular, both informal ones as well as those with formal syntax and semantics, such as UML. Natural language is also used here. On the one hand, the technical concepts of the application view shape the design; they should be clearly recognizable in the design. On the other hand, the design defines the system platform that the software runs on.

2.3 Program View

Here, formal languages dominate programming languages, interfaces of frameworks and libraries (APIs), and communication protocols. The development environment and tools, as well as programming rules, shape the method of working. The high-level program structures are predetermined by the design; in particular, the directory structure of the source code is predetermined by the modularization.

At first glance, the architecture views look like *development phases*. They can be seen as such, but I consider the architecture of a software system primarily in terms of the result—that is, what it is or what it should be—rather than in terms of how it is created. Just imagine, a ready-to-use software system has to be approved before it is used, just like trains, new models of automobiles, or medical devices. The test required for this purpose examines the system's *actual* architecture, based on its documentation and above all how it is manifested in code; it does not examine how the system has been created. In this results-based examination, development processes and procedure models are not important—regardless of whether they are agile or based on phases.

We are repeatedly faced with the differentiation between "rough" and "detailed," such as the differentiation between a rough specification and a detailed specification. This is useful if the rough form is a precise abstraction of the detailed form, but not if rough means only "thereabouts." A rough presentation is also good if it gives a good overview of complex content with a lot of details. However, the application view must in no way be understood as rough, with the design view being a refinement of the application view. Precise details are important in all views.

3 Software Development

Engineering-based software development is of course based on an architecture plan, but in practice there is still a lot of "tinkering" (F.L. Bauer) even today, although this is now referred to as being "agile."

Over the decades, many concepts for processes in software projects have been published and practiced. Various designations based on graphical representations have been defined for the structured process in phases: waterfall model, spiral model, and V model. These metaphors are not significantly enlightening in terms of facts, but they are the object of ideological discussions.

In large organizations, and state organizations in particular, there is a tendency to implement phase and maturity models in a formal, bureaucratic way. The best-known examples are the Capability Maturity Model (CMMI) of the Software Engineering Institute (SEI) in the USA and the German V model. On the other hand, and as a countermovement, a good 20 years ago agile methods came into being, starting with Extreme Programming (XP) and today, primarily Scrum.

Making a plan is an alien concept to proponents of agile; writing down requirements and an architecture design is deemed to be a waste of time and effort. Instead, in a series of steps, mostly referred to as sprints, something is instantly programmed that could be useful to the user. If this gives rise to bungled program structures, refactoring is conducted. That is strange: we do not think about the software structure at the beginning, but when it is ruined after multiple programming sprints, we start to repair it. That is not a structured way of working. There is no engineering-based methodology in the agile methods; consequently, the Scrum guide makes no mention of software engineering.

In his book "Agile!," Bertrand Meyer offers a profound, factual presentation, analysis, and assessment of the agile methods. I have only one thing to add to this, the best thing about the agile methods is their name: agile—a terrific marketing gag. Who dares to say, we are not agile, we work in a structured way?

In a nutshell: regardless of how a team works—with an engineering method, using agile methods, or in some other way—*thought must be given to the software architecture*. It can be designed before programming and written down, or ingeniously hacked into the code—*but the software architecture must have been thought about*. In the end, something is definitely written down—in the code. Hopefully, it works. *In code veritas*.

4 Teamwork

Software systems are complex and large; they can consist of hundreds of thousands or even millions of lines of code. They cannot be developed by one person alone; teamwork is required. This requires structure—in the software, an architecture and in the team, a division of tasks. Not everyone can do everything at any time.

Recently, teamwork has been idealized: mixed groups, with no manager, in flat hierarchies, acting under their own responsibility and self-organizing, they achieve everything easily in daily standup meetings, from sprint to sprint. Deep and thorough contemplation by individuals—for example, about aspects of architecture such as the modularity of a software—is not required. The German Wikipedia article about Scrum states the following: "The development team is responsible . . . for delivery . . . is self-organizing."

Responsibility cannot be ascribed to a team; it can only be ascribed to individual persons. And a professional project team does not come together like a group of friends organizing a party; it is initially put together by someone with personnel responsibility. If there are no further specifications, an informal organization does arise along with leadership, but there are no clear, binding responsibilities. A friend of mine caricatures it with the following saying: "Team = **T**oll, **e**in **a**nderer **m**acht's" (great, somebody else will do it).

In contrast, one of the fundamental experiences of my (professional) life is that people want to and must be led. Of course, the military type of leadership with command and obedience does not come into question for software developers; it is more like that of a football trainer of a professional team or the conductor of a symphony orchestra. A project lead must be convincing and must have the professional skill to instruct employees in both a communicative and a motivational way and to ensure that they (can) achieve good results. On the occasion of the

25th anniversary of the Software Engineering Conference in Garmisch, I wrote something in an Informatik-Spektrum article that still holds true today[3]:

> Good team spirit is more important for the success of a software project than all technology. Therefore, management must constantly strive to ensure conditions for a good atmosphere—a manager must create a good quality of working life. Of course, the atmosphere can only arise in the team itself, with everyone contributing to it; the better the communication and understanding, the better the atmosphere.

5 A Final Wish

Looking back over a quarter of a century of the Software Engineering Prize showed me that a large part of the work is concerned with analytical processes, such as analyzing code and models, but design methods rarely appear. This is regrettable, because software engineering is the doctrine of shaping, designing, building, and developing software systems. Why is there no software engineering school at some universities that states, by way of a standard: This is how you should do it? I would be happy for there to be two or three that compete with one another, possibly with each having a different orientation with regard to the types of systems, be they technical, operational, or media systems.

Is it down to the lack of practical relevance or the short-winded academic life, in which primarily six-page articles are published with the aim of improving the citation index? Textbooks are now rarely written. I would like a constructive software engineering doctrine, as a book, available on paper and of course digitally, supplemented by a website with code examples and further materials.

Most Computer Science graduates who go into business practice work on developing software—for new systems as well as existing ones. Such a doctrine should serve to accompany them on their journey, explaining specifically how to develop. We may then hear in companies, “We follow the X school”—that would be something.

And it could help software engineering, our practical and very relevant discipline, gain standing in the public view again, from which, just like Computer Science, it has disappeared, overrun by the ubiquitous buzzwords.

[3]E. Denert: Software-Engineering in Wissenschaft und Wirtschaft: Wie breit ist die Kluft? (Software Engineering in Science and Business: How Wide Is the Gap?) Informatik-Spektrum, Oct. 1993, page 299.