Preface

Industrial software development is one of the major success stories of the twentieth century. Otherwise, software would not have been able to pervade other areas of life and business, established business models of entire industries would not have been swept away by digitalization, and the global success of Apple, Amazon, Google, Facebook, and eBay would not have been possible.

Software engineering, i.e., the design of larger and larger software systems based on engineering principles, enabled the development of software systems that seemed impossible just a couple of years ago. Therefore, any kind of fundamental denial of this success story is downright absurd (Osterweil et al. 2008). This fact cannot be changed, not even by numerous studies on the alleged state of the software industry, which were in some cases prepared under the flimsiest of conditions, as exposed, e.g., by Eveleens and Verhoef (2010), Glass (2006), or Jørgensen and Moløkken-Østfold (2006).

Yet, time and again, evidence is provided of projects that encounter difficulties—sometimes because the established software development practices have not been followed, sometimes because the individuals involved are too optimistic in their announcements and promises, and in some instances because the numerous individuals involved in a software development project do not have a uniform picture of the actual aim of the project.

It is astonishing that this happens relatively often and is not regarded as a rare exception. Obviously, problems can arise in other projects, not just in software development—airports are finished after serious delays or not at all, public construction projects become more expensive than planned, and trains cannot stop at all platforms. However, genuine project disasters, in the form of a multiplication of the project duration or cost, or in the form of canceled or rolled-back projects, seem to arise more frequently in software development than in other sectors.

Perhaps this is because the immaterial nature of software makes it more difficult to estimate the project state and makes the loss associated with a cancelled project less tangible. Perhaps it is also because software development projects (in which the relevant investment is "only" human resource cost) are often too ambitious and not overly concerned with lean solutions.

Perhaps it is also because the question on the nature of the software process can still not be answered definitively. Is it primarily a production process? Then, it can

be structured from a Taylorist perspective, where detailed specifications are provided, such as in the car production process on an assembly line. Or is it a purely creative process, which is solely driven by the engineer's design talent? In this case, procedural specifications make little sense, in the same manner that the idea of a precise process to create a painting makes no sense. Software engineering seems to lie between these two poles. There are sections that must be clearly regulated and standardized, such as certain testing activities or configuration management. Others cannot be described using algorithms and cannot be supported by a heuristic process method, such as the approach to identify features to be developed at an early stage.

And then there is the phenomenon of uncertainty. Lehman (1989) provided a convincing argument that software projects are exposed to uncertainties; i.e., that during the course of development, situations could arise that were previously unforeseen (or at least uncertain to occur) and for which appropriate support was unknown. Lehman also noticed that, in most cases, these situations could not be identified in advance. Other authors also made this observation early on:

- "Uncertainty is inherent and inevitable in software development processes and products."—Ziv et al.'s uncertainty principle in software engineering (1996)
- "For a new software system, the requirements will not be completely known until after you have a working product."—Humphrey's requirements uncertainty principle (1995)
- It is impossible to fully specify or test an interactive system.—Wegner's lemma (1997)

In light of this finding, which is confirmed in practically every software project, terms such as "software factory" (Cusumano 1989) and titles of scientific articles such as "Software Processes are Software too" (Osterweil 1987) seem misleading or at least ambiguous. Software processes (at least for developing socio-technical systems) are insight-driven processes, they are comprised of more creative than algorithmic parts, and it is certainly the case that they are not precisely foreseeable (Gruhn and Urbainczyk 1998).

This in no way denies the existence of types of software that can be fully described. For example, embedded systems without human interfaces can be completely specified and created in line with the production paradigm.

However, this does not apply for socio-technical systems, for the simple reason that these kinds of systems do not end at the screen, but rather extend into the mind of the user. This does not just mean that software must be prepared for unforeseen user behavior. Rather, in socio-technical systems, the software is only a small part of a system comprised of human and mechanical participants that work together to perform complex processes. This interaction, into which software must seamlessly integrate, cannot be fully described and is also subject to constant change. In particular, when dealing with innovation, with the establishment of new business processes and services, and with the implementation of new automations, the design, implementation, and adaptation of software is a creative process, whose purpose requires continuous calibration. The development of these kinds of software

solutions is not a production process, but rather a cognitive process, which is most likely to succeed when all stakeholders keep an eye on the common goal and pay attention to lean solutions.

Even if these solutions are of a technical nature, the goal they must support is anchored in the application domain and not in information technology (IT). Close communication between enterprise IT[1] and operating departments is unavoidable and essential for success in companies that develop software. However, it is often also characterized by different terminology and, especially, by different types of abstraction (and abstraction capacity).

However, the constant realignment of the project idea, the continuous consultation between enterprise IT and the application domain, and the rejection of the idea of a "software factory" (which suggests a completely predictable software production) also result in a few unpleasant conclusions. For example, the fact that the provision of a complete advance specification is not possible (and that the quest for this is doomed to failure), that there will be late requirements (which only arise during development or even after), that budget allocations and cost estimates are provisional, and that at the start of a project, it is impossible to know precisely what can be obtained and at what cost.

But is this really still necessary? Almost 50 years after the term "software engineering" was coined? After almost 50 years in which the "engineering" in "software engineering" defines a claim, namely the claim of reproducibility, reliability, and calculability? It appears to be so, as software development is still risky, projects still encounter difficulties and, when searching for the causes, the same reasons are constantly identified: a lack of understanding of the application domain, incorrect prioritization, and a lack of communication between the stakeholders (Curtis et al. 1988). Software processes are and will always be cognitive processes, but they must satisfy the expectations of production processes.

Structure and Audience of This Book

This is the challenge that this book deals with—the cognitive nature of software development, the necessity for a unified purpose, the concentration on lean software, the focus on added value, and the omission of the irrelevant. It describes specific instruments and methods enabling all stakeholders to develop a uniform understanding of the software to be created, to determine their genuinely essential requirements, and to deal with changes to this understanding and the requirements.

[1]By "enterprise IT," we refer to a company's enterprise IT department or to external contractors that perform this function.

The **Interaction Room** described in Part II brings all stakeholders together for this purpose—not to a table, but in a room where digitalization and mobilization strategies are jointly developed, where technology potentials are evaluated and where software projects are planned and managed. Why does this require a dedicated room? Because stakeholders can then communicate face to face rather than through e-mails. Because the room can be used to outline complex relationships in a comprehensible manner instead of having to laboriously write them up in great detail. Because there is only room for the most important issues. And because insights are not lost in short-term memory or huge documents, but concisely noted and constantly present. In short, because the Interaction Room makes projects visible and tangible.

The **adVANTAGE contract model** described in Part III ensures that the insight-driven and imprecise process of software development does not just function, but that it is allowed to flourish in a commercial environment, i.e., in a client and contractor relationship. In this model, changes to the project flow are not a reason for stress, but considered normal project events. The contract model ensures that stakeholders focus on generating maximum benefits, creating lean software, and distributing risk fairly despite (or with the aid of) all the changes.

How this can work during the day-to-day running of a project is shown in the **practical example** of the development of an inventory management system for a private health insurance company in Part IV. This is a complex system with, at first glance, an almost unmanageable number of business requirements, statutory conditions, stakeholders, and processes for general and special cases, embedded in the organically developed IT landscape of an insurance company from North Rhine-Westphalia. The example of the project kickoff and the first sprint shows how employees of the company and the IT contractor developed an overview of the project using the Interaction Room, how the design and development was managed, and how efforts were billed.

Ultimately, the success of every single software project, independently of the application domain and the technology used, depends on the **skills** of the stakeholders. Only if the stakeholders are prepared to talk to each other, interact with each other, respect different perceptions of value and effort drivers, reach compromises, pursue innovative solutions, and refrain from political maneuvers, can instruments such as the Interaction Room and adVANTAGE fully unfold their potential. Part V therefore finally describes the requirements profile that software engineers as well as domain experts must satisfy today.

Even though contracting and collaboration may be grounded in two different academic disciplines, they are inseparable in practice where all theory boils down to enabling people to work effectively with each other toward a successful product in a sustainable business relationship.

This book is therefore geared toward CIOs, project managers, and software engineers in industrial software development practice who want to learn how to deal effectively with the inevitable uncertainty of complex projects, who want to

achieve higher levels of understanding and cooperation in their relationships with customers and suppliers, and who want to run their software projects at lower risk despite their inherent uncertainty.

Acknowledgments

The authors would like to thank Simon Grapenthin for sharing his extensive hands-on experience in facilitating Interaction Room workshops and training Interaction Room coaches in a wide range of business domains. We would also like to thank Sandra Delvos for countless hours of designing and revising the book's illustrations, and Alexander Lohberg and Anja Wintermeyer for their background research.

Reykjavík, Iceland — Matthias Book
Essen, Germany — Volker Gruhn
Berlin, Germany — Rüdiger Striemer

References

Curtis B, Krasner H, Iscoe N (1988) A field study of the software design process for large systems. Comm ACM 31(11):1268–1287. doi:10.1145/50087.50089

Cusumano MA (1989) The software factory: A historical interpretation. IEEE Software 6(2): 23–30. doi:10.1109/MS.1989.1430446

Eveleens JL, Verhoef C (2010) The rise and fall of the Chaos report figures. IEEE Software 27(1):30–36. doi:10.1109/MS.2009.154

Glass RL (2006) The Standish report: Does it really describe a software crisis? Comm ACM 49(8):15–16. doi:10.1145/1145287.1145301

Gruhn V, Urbainczyk J (1998) Software process modeling and enactment: An experience report related to problem tracking in an industrial project. In: Katayama T, Notkin D (eds) ICSE'98: Proc 20th Intl Conf Software Engineering, pp 13–21. doi:10.1109/ICSE.1998.671098

Humphrey WS (1995) A discipline for software engineering. Addison-Wesley, p 349

Jørgensen M, Moløkken-Østvold K (2006) How large are software cost overruns? A review of the 1994 Chaos report. Information and Software Technology 48(4):297–301. doi:10.1016/j.infsof.2005.07.002

Lehman MM (1989) Uncertainty in computer application and its control through the engineering of software. J Software Maintenance 1(1):3–27. doi:10.1002/smr.4360010103

Osterweil LJ (1987) Software processes are software too. In: Riddle WE (ed) ICSE'87: Proc 9th Intl Conf Software Engineering, pp 2–13

Osterweil LJ, Ghezzi C, Kramer J, Wolf AL (2008) Determining the impact of software engineering research on practice. IEEE Computer 41(3):39–49. doi:10.1109/MC.2008.85

Wegner P (1997) Why interaction is more powerful than algorithms. Comm ACM 40(5):80–91. doi:10.1145/253769.253801

Ziv H, Richardson DJ, Klösch R (1996) The uncertainty principle in software engineering. Technical Report UCI-TR-96-33, University of California, Irvine. http://www.ics.uci.edu/~ziv/papers/icse97.ps. Accessed 23 Feb 2016

http://www.springer.com/978-3-319-41476-8

Tamed Agility
Pragmatic Contracting and Collaboration in Agile Software Projects
Book, M.; Gruhn, V.; Striemer, R.
2016, XVI, 334 p. 66 illus., 20 illus. in color., Hardcover
ISBN: 978-3-319-41476-8