Formalisms in Software Engineering - Myths versus Empirical Facts

Dieter Rombach and Frank Seelisch
[dieter.rombach, frank.seelisch]@iese.fraunhofer.de

Fraunhofer Institute for Experimental Software Engineering

Abstract. The importance of software grows in all sectors of industry and all aspects of life. Given this high dependability on software, the status of software engineering is less than satisfactory. Accidents, recall actions, and late projects still make the news every day. Many of the software engineering research results do not make it into practice, and thereby the gap between research and practice widens constantly. The reasons for not making it into practice range from isufficient commitment for professionalization of software development on the industrial side, to insufficient consideration for practical scale-up issues on the research side, and a tremendous lack of empirical evidence regarding the benefits and limitations of new software engineering methods and tools on both sides. The major focus of this paper is to motivate the creation of credible evidence which in turn will allow for less risky introduction of new software engineering approaches into practice. In order to overcome this progress hindering lack of evidence, both research and practice have to change their paradigms. Research needs to complement each promising new software engineering approach with credible empirical evidence from in vitro controlled experiments and case studies; industry needs to baseline its current state of the practice quantitatively, and needs to conduct in vitro studies of new approaches in order to identify their benefits and limitations in certain industrial contexts.

Keywords: computer science, [empirical] software engineering, software development, empiricism, empirical evidence

1 Introduction

Software plays an ever increasing role in all aspects of our lives. The mere amount of code embedded in modern, highly integrated products should mandate that the development of software follow widely agreed principles. Furthermore, these principles ought to ensure quality goals that have been specified in advance. The function of software in life supporting systems as well as in numerous standard applications which allow to perform business processes more efficiently, proves that software has become vital in all respects.

However, we witness serious system failures resulting from faulty software, sometimes with tremendous consequences: People die, assets are being lost, and products need to be recalled by renowned OEMs. Software engineering today, seen as a practically highly relevant engineering discipline, is not mature enough considering the role it plays.
Likewise, the relationship between computer science and software engineering needs to be stated more precisely and lived, so that theoretical research in computer science can be translated into practical methods and procedures that can be utilized by software engineers in software development projects.

The major claim of this work is that typical shortcomings in the practical work of software engineers as we witness them today, result from missing or unacknowledged empirical facts. Discovering facts by empirical studies is the only way to gain insights in how software development projects should be run best, i.e., insights in the discipline of software engineering. Empirical facts will in turn motivate computer science research.

This paper is organized as follows. The following Chap. 2 discusses the role of software in industry and society, and adds an economic standpoint. How do companies estimate the importance of software for their business? What are proven economic guidelines for realizing software development projects?
Chapter 3 summarizes the typical practical problems resulting from immature software engineering, and identifies the shortcomings of software engineering in practice. It comes up with general explanations for the problems we are currently confronted with when developing software in critical settings, e.g., within tight schedules. Moreover, we argue that the current situation is likely to become even more critical, as software systems become more and more complex.
Having detailed practical problems of software engineering, Chap. 4 addresses the possible solutions to these problems offered by research. Starting with general principles of computer science we go top-down towards software engineering as its practical toolbox for developing software, and finally to the most important ingredient of software engineering: empirical facts.
Chapter 5 illustrates the need for more empirical software engineering, both generally and by means of the concrete example of the special inspection technique *reading*.
We close with an outlook on next steps.

2 The Role of Software in Industry

In the previous section, the spectrum of common problems due to software failures has been detailed. From an economic standpoint, these problems imply direct costs as well as great efforts for repairing, fitting, bug-fixing and necessary changes in software versions, products, and processes. In order to save precious resources, *companies need to enforce the usage of best practice methods and procedures of software engineering.*

On the other hand, software engineering has been offering and still offers completely new ways for developing products.

2.1 Software as Driver for Innovation

Due to its enabling role, software has become a major factor in today's industry. Industrial leaders in the automotive business, in the field of medical devices, and logistics estimate that 80% of all innovation is directly triggered by software.
However, most non-IT companies do not reflect these estimates in their organizational structure: Software is important but IT sub-organizations currently do not seem to be. It is a common model to have these sub-organizations raise their funds within the company instead of being given basic funding.
We expect that this organizational setup adds to the current problems in ongoing software projects, as strategic planning cannot be sufficiently considered.

2.2 Economic Aspects of Software Development

In the context of globalization, industrial companies focus more than ever on the goal parameters *quality, cost,* and *time to market*. Especially quality promises to offer better chances to establish unique selling propositions in the lucrative upper market segments. For most Western economies, this may offer a way to legitimate the much higher level of salaries compared to Asian economies.
In terms of software engineering, this means that development projects will also primarily be managed and steered by these parameters. And especially concerning the parameter *time to market* companies face a classical trade-off: Is it more important to deliver a new software version fast, while taking into account that more effort will have to be spent on bug-fixing? Or, is it the goal of the company to minimize costs over the entire

software life cylce?

According to a model for business development by Stalk, Evans and Shulman [1] companies need to adher to the following, very roughly sketched strategic roadmap:

1. They need to be clear about their overall strategic goals.
2. They need to identify supporting business processes that guarantee the given strategic goals.
3. They need to prioritize the most important sub-processes and continually invest into them.

In the context of software development, the business processes are software development processes. The highest priority processes to be invested into in the case of safety goals, could be design and verification processes. Investment into a software engineering subprocess P means to invest into the creation of empirical evidence regarding its effectiveness f with respect to some goal G in the context of the given environment C. G could be any one of the afore mentioned goals quality, cost, or time.

Thus, the challenge could be phrased as identifying the following function:

$$G == f(P, C), \tag{1}$$

where "==" stands for an empirically based relationship.

As a consequence of the processes defined under 2. and the selection made under 3., a number of software development projects with management attention will normally be started. Note that the above question whether a software version should be delivered quickly or whether the company should aim at minimizing total cost along the product life cycle, is *not* answered by the above generic roadmap. From an IT perspective, this decision has implications for software attributes such as adaptability and sustainability: The former strategy will in general lead to a higher adaptability, whereas the latter naturally leads to more sustainable software solutions.

3 Practice of Software Engineering

3.1 Problems

Today's software engineering practice is mainly characterized by the following two phenomena. **Schedule and Budget Overrun:** Taking a

closer look at software development projects in industrial settings reveals that schedule and budget overrun is not at all uncommon. In some projects, rates of schedule overrun of up to 150% have been witnessed.

Safety Criticality: Besides these mere management obstacles, accidents with sometimes dramatic consequences are a much more serious problem. Generally speaking, here, software failures imply safety-critical situations while handling products with embedded software, e.g., cars, trains, or airplanes. Whenever serious safety problems had been detected, OEMs had to recall their products, which typically results in a great loss of assets both economically and in terms of company reputation and product image.

3.2 Reasons for Current Problems

Non-Compliance with Best-Practice Principles of Software Development such as:

- encapsulation,
- information hiding,
- proven architectural patterns,
- traceability, e.g., diversion of documentation versus code over time.

The year 2000 problem (2YK) is a prominent example of poor encapsulation and information hiding. Generally speaking, information hiding will lead to small interfaces between program modules. This does not only increase readability and manageability of code but also enables a potentially higher reuse of these modules, most likely at a lower cost.

Non-Compliance with Best Practice Principles of Process Design such as:

- review and inspection techniques for an earlier defect detection; see [2] and Sect. 5.2,
- best practice process patterns,
- best practice process design tools, e.g., Waterfall or V-Model.

Non-Existence of Credible Evidence Regarding the Effects of Methods and Tools, i.e.,

- missing empirical facts,
- missing context information,

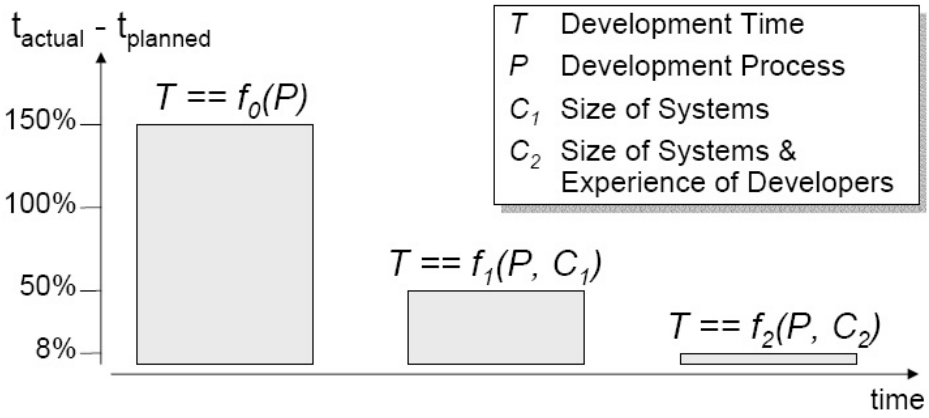

Fig. 1. Relationship between Context Understanding and Predictive Capability

– missing certainty information.

In most problematic software development projects, teams will have to deal with a combination of both kinds of non-compliance, and will additionally suffer from missing context information, or empirical facts.

Figure 1 refines formula (1) given in Sect. 2.2. It illustrates how the precision of the prediction of project development time T for a process model P may depend on context C: Without any context knowledge, prediction across projects may be off by as much as 150%; see [3]. By taking into consideration parameters such as size of the system to be developed and experience of the developers, one can reduce the variance significantly; in many cases to a single-digit variance.

3.3 Future Trends & Challenges

The outlined current situation is not likely to improve by itself. Actually, there are some trends which will most probably aggravate the setting.

Embedded Systems and Increasing Complexity: More and more software is being embedded in life-supporting systems, e.g., medical devices used in operation theatres. Software is key to realizing new functions without including too many additional hardware components. Thereby, systems are becoming more and more complex. We have begun to build highly integrated *systems of systems* in which the same piece of software implements more than just one system function.

Cross-Disciplinary Systems: Software systems find their way into domains that have previously been dominated by classically engineered solutions. Here, classical disciplines and software engineering already begin to form new hybrid disciplines for which only few skilled engineers are available.

Ubiquitious Computing and Ambient Intelligence: With ubiquitious computing and ambient technologies we witness a miniaturization combined with a remarkable multiplication of new, spontaneously connecting components. These concepts drive the development of highly flexible, service oriented software architectures which ensure high degrees of robustness.

Loss of Direct Control: Ubiquitious Computing and Ambient Intelligence will also push the development of globally interconnected information systems, and systems with autonomous control that are able to spontaneously establish networks and reorganize themselves. The human user will thus no longer be able to directly control these systems.

It is hard to forecast what in detail these trends are going to imply with respect to the development of software.
Most likely, development standards will need to become more definite and resilient, in order to ensure that software operate as specified and be safe, secure, and trustworthy.

4 Research in Software Engineering

4.1 Computer Science & Software Engineering

Computer science is the well-established science of computers, algorithms, programs, and data structures. Just like physics, its body of knowledge, can be characterized by facts, laws, and theories. But, whereas physics deals with natural laws of our physical world, computer science is a body of *cognitive laws*; cf. [4]
Additionally, when software engineering deals with the creation of large software artifacts, then its role is more similar to mechanical and electrical engineering where the goal is to create large mechanical or electronic artifacts.
Figure 2 shows the positioning of computer science and software engineering in the landscape of sciences.

Each science provides a kernel set of principles that gives rise to a set of practical methods for manipulating the world in one or the other useful

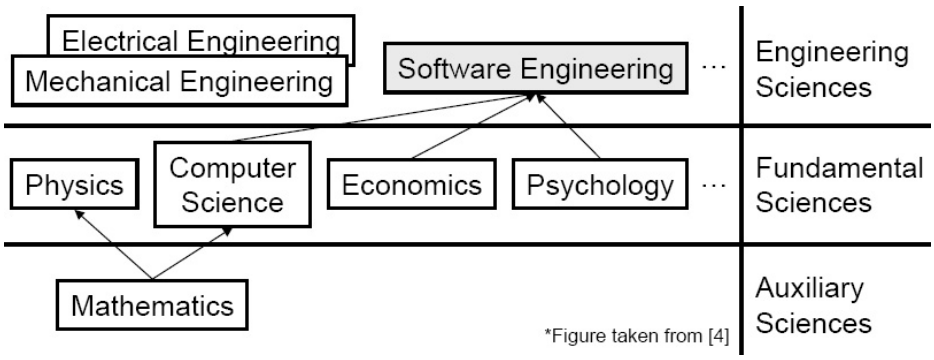

Fig. 2. Positioning Software Engineering in the Landscape of Sciences

way. For example, civil engineering methods including statics calculations can be applied to build a bridge; here the kernel set of laws to do so correctly stems from mathematics and physics.
In this sense, software engineering can be seen as the analog set of methods for developing software, based on fundamental results from computer science. For instance, research in computer science gave rise to functional semantics. From that formal foundation, software engineering derived inspection techniques such as *stepwise abstraction* and *cleanroom development*, and proved furthermore the practicability of these methods by means of empirical studies.
Software engineering needs to be based upon formal foundations, but on the other hand, pure computer science concepts are generally not applicable in practice, e.g., due to algorithmic complexity.

4.2 Software Engineering Principles

Let's take a look at a very general and powerful principle of computer science:
If we need to solve a hard problem, we may try to partition it into smaller subproblems for which we know how to solve them. The basic recursive algorithm for this general pattern of *divide and conquer* is shown in Fig. 3. (Problem partition and combination of partial solutions will of course depend on the particular problem at hand.)

Divide and Conquer in Software Development Projects We also use this *divide and conquer* principle when working in software devel-

```
solve_problem(Problem p)
  if algorithm_available_for(p) then
    a ⟵ algorithm_for(p)
    s ⟵ apply_algorithm(a, p)
  else
    {p_i : i ∈ I} ⟵ partition_of(p)
    for each i ∈ I
       s_i ⟵ solve_problem(p_i)
    s ⟵ combine_solutions({s_i  i ∈ I})
  end if
  return s
```

Fig. 3. Divide and Conquer - Solving a Difficult Problem by Partitioning

opment projects: If the software development process is large, software engineers typically try to partition it into subprocesses with well-defined milestones.
If the task is to create a software product for performing a variety of related business tasks, they attempt to partition the set of requirements into subsets with small mutual overlap.

The following insights have been extracted from a series of experiments:

- In large projects with comparably low risk, e.g., due to available domain knowledge and an experienced project team (see Fig. 1), it is best to use process-oriented development models, like e.g. the Waterfall or V-Model.
- Is the project small but characterized by a high risk, e.g., due to missing domain knowledge and thus the necessity to apply a general approach, software engineers should apply product-oriented models, e.g., agile development methods.
- Large projects with high risk resulting from tight deadlines and poor domain knowledge are best run using product-oriented models, e.g., incremental development techniques.

(The fourth remaining case - small projects with low risk - are the ones that normally pose few problems. Moreover, they are irrelevant for most real-world settings.)

Only when the above software engineering principles will be adhered to, software development teams will have a chance to manage software development projects according to pre-defined budget and schedule.

4.3 Empirical Evidence

The key to success in software development projects is to define measurements and consequently use them in order to be able to measure work progress and detect failure or accomplishment in a rational and transparent manner.
That means that milestones in a development project need to be clearly marked so that responsible and further involved people can determine at any time whether they have been reached or not.
Note that having definitions and measures at one's disposal does not automatically guarantee that they be used. Inforcing their usage must be part of the project and can often only be accomplished by organizational changes or even changes in the working culture. This, in turn, requires top management commitment.

4.4 Evidence is Context-Dependent

The challenge in software engineering, as a practical standard method and tool box for the development of complex software, is that most tasks will typically have to deal with organizations and will have to address human requirements of some kind. This has two concequences:

1. The methods will vary from organization to organization. Thus, software engineering will depend on the environment in which it is being applied; that is, it is context-sensitive; see again Fig. 1.
2. State of the art software engineering will change over time, as technical progress takes place, and working culture evolves.

An important consequence of both items is that we need to clearly document in what context C a software engineering method can be applied with what result, that is, we need to document the nature of f in formula (1) of Sect. 2.2. Furthermore, a mechanism for periodically revising our knowledge is required, in order to have a valid set in place at any time.

Laying the foundations of software engineering thus means to

- state **working hypotheses** that specify software engineering methods and their outcome together with the **context** of their application,
- make experiments, i.e., studies to gain **empirical evidence**, given a concrete scenario,
- formulate **facts** resulting from these studies, together with their respective context,

– abstract facts to **laws** by combining numerous facts with similar, if not equal, contexts,
– verify working hypotheses, and thereby build up and continously modify a concise **theory** of software engineering as a theoretical building block of computer science.

Current problems of software engineering in pratice can be directly related to these goals:

– Clear working hypotheses are often missing.
– There is no time for, or immediate benefit from empirical studies for the team who undertakes it.
– Facts are often ignored, or applied in differing contexts. Moreover, facts are often replaced by myths, that is, by unproven assumptions.
– Laws are rarely abstracted from facts. The respective contexts are sometimes equated which will lead to false laws; cf. examples in Sect. 5.1.
– Up to now, a concise, practicable theory of software engineering does not exist.

5 Empirical Software Engineering

Let us come back to empirical evidence as the most important means to fill gaps in the body of knowledge of software engineering, cf. Sect. 4.3. and [5]. Methods and tools for performing empirical studies exist. Nevertheless, we still do have a lot of myths which impact our discipline in a negative way. Section 5.2 elaborates the example of reading-based inspections vs. testing, to demonstrate how proper use of empirical methods can turn the myth that *"testing is more effective than reading"* into a law that *"in general, reading is more effective"*.

Figure 4 illustrates that there is no software engineering other than empirical software engineering: Its technical building blocks - *formalism, systems*, and *processes* - need to be based on empiricism which rests, in turn, on computer science and mathematical foundations.

Prominent representatives of the existing empirical tool box are:

– GQM: *Goal Question Metrics* support decision making in order to guide the software development team towards the relevant measurements.
– QIP: *Quality Improvement Paradigm* have a strong empirical focus.
– EF: *Experience Management* deals with experience and hence expertise in project teams, preferably across different domains.

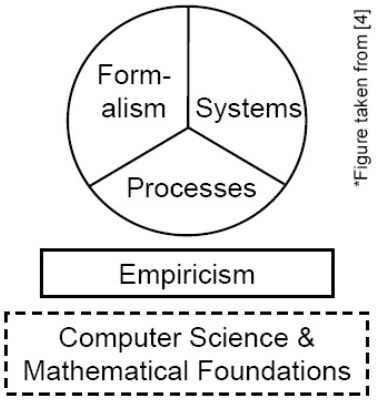

Fig. 4. Categories of Software Engineering

The Software Engineering Lab (SEL) of NASA's Goddard Space Flight Center has been the first organization establishing a sound experience management along-side a practical software development unit. They have achieved significant and sustained improvements over the years; cf. [6]. For these accomlishments, the SEL had been the first recipient of Carnegie Mellon's Software Engineering Institute's (SEI) Software Process Achievement Award.
According to [7], today, the top institutes on empirical software engineering research are:

- Fraunhofer IESE + CESE, Germany and USA, respectively,
- Simula Research Lab, Norway, and
- NICTA Empirical Group, Australia.

5.1 Facts versus Myths

Today, empirical software engineering is best characterized by the following three statements:

1. If facts are missing then this gives rise to myths.
2. Concerning software development projects, there are more facts available than being used. Often, software engineers ignore available facts. (This, in turn, often happens under the pressure of unrealistic project schedules.)
3. Besides ignored empirical facts, there exist indeed many gaps of empirical knowledge.

Here are some examples for unproven hypotheses which give rise to myths in empirical software engineering:

- *"Changes are easier when made earlier in the software development process."*
- *"Pair reviews are effective and efficient."*
- *"Re-factoring replaces design for modifiability."*
- *"As the cost of defect reduction increases with lag time, does this mean that we need to focus more on inspections and reviews?"*

Likewise, the following trade-offs are still unresolved and support the insistence on related myths:

- *global distribution of software development versus local concentration*
- *subcontracting versus in-house projects*
- *construction for reuse versus one-time construction*
- *large teams versus small teams (resulting in a longer development time)*

We add some open research questions resulting from established laws; in the sense of the defintion given in Sect. 4.4. We will only be able to answer those based on new empirical evidence. (The cited laws result from the work of Boehm, Endres, Basili, Selby, and Rombach, et. al.)

- Law: The cost of defect reduction increases with lag time. Question: *Does this mean that we need to focus more on inspections and reviews, or rather on the design of easily modifiable systems?*
- Law: Formal reviews reduce cost of rework and thus total development effort. Questions: *Under what conditions do object-oriented techniques reduce development effort? Under what conditions does commercial-off-the-shelf (COTS) software reduce development effort?*
- Question: *What is the relationship between good designs and domain knowledge?*
- Question: *Under what conditions can changes be implemented at less cost by means of agile methods?*
- Law: For software components, inspections and reviews are more effective and efficient than testing. (As the law states, this has been proven for software components; initially by Basili and Selby, see [8], and sustained by many other studies.) Question: *Is this also true for entire systems?*

We close this paragraph by giving a concise example of an empirical study. This example is to serve as a guideline for setting up a typical software engineering experiment. It dealt with the special inspection technique *reading*.

5.2 Example: Reading

Reading has become a key engineering technique in the toolbox of inspection procedures. It supports the individual analysis of any textual software document that may be dealing with requirements, design, code, test plans, etc. Generally speaking, reading enables local improvements in the software development process, implying global effects.

The experiment [9] provided insight into the effect of different variables, such as experience of readers and type of defects, on the reading technique. It included

- the investigation of early code reading versus testing experiments,
- the introduction of reading into NASA's cleanroom process,
- the replication of experiments and results in other groups, and
- the transfer of the results in other industries.

The results showed that reading

- can reduce failure rates by 25%,
- finds 90% of faults before testing,
- increases productivity by 30%,
- helps to better structure code in future projects, based on learning from reading,
- increases the predictability of project performance parameters such as cost, and compliance with schedules.

6 Conclusions & Outlook

In the previous chapter, we argued that empirical studies are the key to filling gaps in our knowledge of the field of software engineering. Only empirical evidence can give rise to facts and new laws. Consequently, there can be no software engineering other than empirical software engineering.

In order to address the most relevant research questions, a revision of agendas is necessary; especially

- the research agenda,
- the practical empirical agenda, i.e., we need to plan what experiments need to be performed in what contexts; ideally coordinated by a research network such as the International Software Engineering Research Network (ISERN), see [10], and
- the educational agenda.

The last item addresses the need to educate researchers for whom software engineering is naturally build upon an empirical foundation, and for whom experiments are the standard means to do research in software engineering.

Industrial organizations need to adopt long-established, well-founded engineering methods for the development of safe, secure, and trustworthy software. Concerning the project management side, they need to accept planning and working schemes, including schedules, that have been defined by experienced computer scientists and software engineers. Experience guarantees a bounded variance of software development time.

Last but not least, IT sub-organizations inside non-IT companies need to be granted a better standing, in order to function as a natural anchor for software development projects.

References

1. Stalk, G., Evans, P., Shulman, L.: Competing on Capabilities: the New Rules of Corporate Strategy. Harvard Business Review (March-April 1992) 57–69
2. Boehm, B., Basili, V.: Software Defect Reduction Top 10 List. Computer **34**(1) (2001) 135–137
3. The Standish Group: The Chaos Report 1994. World Wide Web (1994) http://www.standishgroup.com/sample_research/PDFpages/chaos1994.pdf.
4. Broy, M., Rombach, D.: Software Engineering. Wurzeln, Stand und Perspectiven. Informatik Spektrum **25**(6) (December 2002) 438 – 451
5. Endres, A., Rombach, D.: A Handbook of Software and Systems Engineering. Addison-Wesley Longman, Amsterdam (May 2003) ISBN-10: 0321154207, ISBN-13: 978-0321154200.
6. Basili, V., Zelkowitz, M., McGarry, F., Page, J., Waligora, S., Pajerski, R.: Special Report: SELs Software Process-Improvement Program. IEEE Software **12**(6) (November 1995) 83–87
7. Ren, J., Taylor, R.: Automatic and Versatile Publications Ranking for Research Institutions and Scholars. Communications of the ACM **50**(6) (2007) 81–85
8. Basili, V., Selby, R.: Comparing the Effectiveness of Software Testing Strategies. IEEE Transactions on Software Engineering **13**(12) (December 1987) 1278–1296
9. Basili, V., Green, S.: Software Process Evolution at the SEL. IEEE Software **11**(4) (1994) 58–66
10. ISERN: International Software Engineering Research Network. World Wide Web http://isern.iese.de/network/ISERN/pub/.